Titre: Title:	Nanowalker software system development
Auteur: Author:	Jing Yang Zhang
Date:	2006
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Zhang, J. Y. (2006). Nanowalker software system development [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7843/

UNIVERSITÉ DE MONTRÉAL

NANOWALKER SOFTWARE SYSTEM DEVELOPMENT

JING YANG ZHANG
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2006

Your file Votre référence
ISBN: 978-0-494-25586-5
Our file Notre référence
ISBN: 978-0-494-25586-5

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

NANOWALKER SOFTWARE SYSTEM DEVELOPMENT

présenté par: ZHANG Jing Yang
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. BOYER, François-Raymond, Ph.D., président
M. MARTEL, Sylvain, Ph.D., membre et directeur de recherche
Mme NICOLESCU, Gabriela, Doct., membre

Acknowledgment

This thesis was conducted in the Department of Computer Engineering at Ecole Polytechnique, University of Montreal. I would like to thank my supervisor, Professor Sylvain Martel, for giving me the opportunity to do the research in the NanoRobotic Laboratory, the encouragement, and support.

I appreciate the cooperation of Marc-Antoine Fortin, Pascal Hannoyer, Dominic St-Jacques, Hong Jun Song, and other people who worked together with on the NanoWalker project. I am also grateful for the help I have gained from all members in the NanoRobotic Laboratory.

In addition, I would like to thank Neila Kaou and Kwang Soo Kim who gave me valuable advices on writing this thesis.

Finally, I thank my wife and all of my family members for their support.

Résumé

Avec l'invention du microscope à effet tunnel et les autres équipements de haute résolution, la nanotechnologie permet d'ouvrir la porte d'un nouveau monde microscopique. Ces découvertes ont aussi apporté beaucoup de défis dont l'un des plus important, la communication moléculaire.

Cette thèse de maîtrise est centrée sur le développement d'un logiciel pour coordonner une flotte de robots miniatures (NanoWalker). L'objectif du projet NanoWalker est de positionner un certain nombre de robots à des sites avec une précision d'une centaine de nanomètres et de permettre à ces robots de faire des tâches au niveau subatomique. Un prototype de ce système a déjà été créé par d'autres chercheurs. L'objectif de recherche pour cette thèse est de continuer le développement de l'architecture logiciel qui supporte la plate-forme puisque il y a tout un monde entre le prototype et le monde réel. De nombreux problèmes devront d'abord être résolus. La majeure partie de cette thèse et de ce projet de recherche est de développer un canal de communication infrarouge entre les robots et la plate-forme de contrôle.

Un NanoWalker est un robot miniature qui peut se déplacer librement sur la plate-forme de travail. La faible dimension du Nanowalker limite grandement sa puissance de calcul. La majeure partie du traitement de donnée est faite par la plate-forme de contrôle principale, la Central Control and Management System. Les données, entre le NanoWalker et le Central Control and Management System, sont échangées à travers un canal de communication infrarouge. Le protocole de communication est le Fast Infrared protocol qui permet un taux de transmission de 4Mb/s. Les éléments hardware et logiciel impliqués dans la communication infrarouge forment la canal de communication infrarouge.

Le canal de communication emploi une méthode de partage du temps pour allouer des espaces de temps à un certain nombre de robot de façon circulatoire, pour pouvoir contrôler plusieurs robots en même temps. Comment allouer l'espace de temps aux robots efficacement et fidèlement est la difficulté majeure pour le développement de ce canal de

communication. L'efficacité de l'allocation a pour but d'optimiser l'utilisation de la largeur de bande de 4Mb/s et ainsi de minimiser les délais et latences.

Plusieurs éléments ont été complété durant ce travail de recherche: 1)conception d'une nouvelle architecture à six niveaux contrôlant et coordonnant un certain nombre de robots et leur environnement de travail; 2) conception d'une nouvelle structure de donnée des paquets de communication; 3) conception et implémentation d'un module de gestion de la communication infrarouge; 4) conception et implémentation du logiciel interne au Nanowalker gérant la communication infrarouge; 5) test du canal de communication.

Abstract

With the invention of the Scanning Tunneling Microscope and other high resolution equipment, Nanotechnology enables us to open the door to a new micro-structure world. It also brings much challenge, such as the molecular communication, one of the major challenges.

This Master's thesis focuses on the research of developing the software controlling system for coordination of the large number of miniature robots (NanoWalker). The objective of the NanoWalker project is to position a number of miniature robots to targets with a hundred-nanometer-scale precision and allow these robots to do some sub-atomic scale works. The prototype model of this controlling system has already been created by some other researchers. The goal of the research for thesis is to continue the development of the software infrastructure for the supporting platform since from the prototype to real implementation, numbers of problems should be solved first. The majority of the research work concentrates on developing the infrared communication channel between the supporting platform and the individual robots.

A NanoWalker is a miniature robot which can move freely on a working platform. The dimension of the NanoWalker greatly limits the data processing ability of the robot. Most of the computational works are done on the main controlling platform, namely the Central Control And Management System. The data exchanged between the NanoWalker and the Central Control And Management System is done through a infrared communication link. The infrared communication protocol is the Fast Infrared protocol which has 4Mb/s transmission rate. The hardware components and the software components involved in the infrared communication link form an infrared communication channel.

The communication channel uses a time sharing method to allocate time slots to a number of robots rotationally, so that it can control multiple robots at the same time. How to allocate time slots to the robots efficiently and reliably is the major concern in developing the channel. The allocation efficiency is to ensure that the 4Mb/s transmission

bandwidth is fully utilized, so that the data transmission latency is minimized.

A few things have been achieved during the research of this thesis: 1) redesigning the six-layer distributed architecture that controls and coordinates a set of robots and their working environment; 2) redesigning data communication package structures; 3) design and implement the infrared communication manager module; 4) design and implement the infrared communication handling part of the programming running inside the NanoWalkers; 5) test of communication channel.

Condensé en français

Le projet NanoWalker (NW) a été inauguré par le Laboratoire de Bio-Instrumentation de L'**Institut Technologique au Massachusetts** (MIT), sous la direction du professeur Ian Hunter et du professeur Sylvain Martel. Le but du projet est de construire une flotte de robots nommés NanoWalkers permettant de détecter et de manipuler simultanément les échantillons mécaniques ou biomédicaux à l'échelle moléculaire et/ou atomique.

La recherche est concentrée sur le développement du logiciel pour le système de NanoWalker, et l'objectif de la recherche implique principalement les aspects suivants:

1. Continuer la conception de l'architecture du système de contrôle, et faire la modification nécessaire sur la conception précédente.

2. Concevoir et appliquer le Manager de Communication **Infrarouge** (IR), et, de plus, assurer le canal de communication IR entre le **système de contrôle et de destion central** (CCMS) et les robots afin qu'ils soient prêts pour l'intégration.

3. Essayer de rendre l'intégration possible pour les parties achevées dans le système de contrôle, et découvrir n'importe quels problèmes dans les propositions de leur conception par une série de tests.

A.

Un NW est un robot miniaturisé sans fil, élaboré pour les opérations à l'échelle du nanomètre, avec d'un **microscope à effet tunnel** (STM) comme son actionneur. Le projet du NW vise à construire une plate-forme composé d'un grand nombre de robots qui travaillent simultanément permettant ainsi l'obtention d'une plate-forme à haut rendement. Tous les robots dans le système sont contrôlés par le CCMS avec les liaisons IR. Chaque robot peut se déplacer librement sur la plate-forme de travail, et est propulsé par le biais de trois tubes piézo-électriques. Le système de positionnement des robots est la combinaison d'un **système de positionnement globale** (GSPS) et d'un **système de positionnement**

atomique (ASPS). Le GSPS utilise un **Position Sensing Device** (PSD) et le signal de positionnement IR pour identifier la position de chaque robot. Chaque liaison IR est responsable de transmettre les données de température internes et les données de STM d'un NW au CCMS, et de transmettre les données de mouvement des pattes arrières.

La difficulté majeure rencontrée lors de la fabrication du NanoWalker est la limitation de sa grandeur. En effet, la surface du circuit imprimé est trop petite pour mettre plus de composantes électroniques. On ne peut donc pas espérer qu'un NW comporte de l'intelligence artificielle qui lui permettrait de travailler indépendamment. Le seul composant pour traiter les données est le **processeur de signal numérique** (DSP), qui a une très petite capacité de traitement des données. Cette limitation détermine que le CCMS doit effectuer l'essentiel du travail de gestion des robots.

Le CCMS est compose d'une série d'**ordinateurs** (PCs). Chacun est responsable d'un aspect différent des tâches de contrôle. Ces ordinateurs exécutent leurs tâches sur les **systèmes d'exploitation** (OS) différents. La figure 2-7 représente les connexions du réseau dans le projet du NW. À présent, l'implémentation utilise quatre PCs pour contrôler des robots sur la plate-forme de travail. Un PC est utilisé pour traiter le GSPS et un autre est responsible du traitement de gestion de communication IR. Les deux PCs sont responsables de contrôler une cellule de travail. Ils forment la plate-forme sur laquelle le gestionnaire de robot fonctionnera.

B.

Le système logiciel du CCMS est le **Système de Contrôle de la Plate-forme du NanoWalker** (NWPCS). Il est responsable du contrôle du système du NW. Il n'inclut pas le programme de gestion fondamentale des I/O qui œuvre dans le DSP du NW. Le NWPCS doit pouvoir répondre aux changements du système entier en temps-réel, et les résultats de ses réponses doivent être fiables. D'ailleurs, à l'étape actuelle, la plupart des composantes de l'architecture matérielle ne sont pas déterminées. C'est pourquoi, l'architecture logicielle doit être indépendante et flexible aux changements de spécifications des composantes matérielles.

L'architecture originelle proposée par les autres étudiants a six niveaux. En commençant par la plus basse, les niveaux sont: le niveau matériel, le niveau de la driver (d'appareil), le niveau manager, le niveau agent, le niveau NonaOS, et le niveau application. Selon la figure 3-1, le niveau agent inclut l'Agent Environnement et l'Agent Robot. L'Agent Environnement contrôle le Manager DAQ et le Manager Chambre. L'Agent Robot manipule le Manager Communication et le Manager Position. Tous les managers sont responsables pour administrer leurs drivers et des composantes matérielles. Les composantes logicielles de chaque niveau sont indépendantes l'une de l'autre. S'ils veulent partager quelques pièces d'information, l'information partagée doit passer par leur niveau supérieure. L'information à travers chaque niveau sera distribuée par la façon de le communications inter-niveaux, qui, en fait, constituent les liens de communication du réseau TCP/IP.

La conception originelle a quelques problèmes conceptuels qui ont besoin d'être réglés avant de décrire plus de designs de l'architecture en détail:

- le niveau matériel n'est pas nécessaire. Il est remplacé par le programme qui fonctionne dans le corps du NW, qu'on baptise le **NanoWalker OS Fondamental** (NWBOS).

- Dans le niveau driver, chaque driver d'appareil devrait être développé au point où aucune opération qui soit pertinente au matériel de système a besoin d'être soigné par les régisseurs. Dans la proposition originelle, le caractère fonctionnel de ce niveau n'a pas été mentionné clairement. Les drivers d'appareil doivent fournir des chemins pour accéder à le niveau supérieure, et établir les registres et les paramètres I/O dans les composantes matérielles selon les usages.

- Au lieu de transmuter les données entre le niveau agent et le niveau driver, le rôle de le niveau manager est de coordonner les drivers et d'assurer que les agents obtiennent des données crédibles du système NW et les échantillons.

- L'architecture doit permettre le contrôle des multiples cellules de travail.

- Le concept de communication 'inter-niveau' devrait être remplacé avec le concept 'inter-ordinateur'.

En se basant sur les nouvelles considérations mentionnées ci-dessus, une architecture révisée a été proposée pour résoudre les problèmes susmentionnés, comme illustré dans la figure 3-2.

Les communications 'inter-ordinateur' dans le système NW sont achevées en utilisant les **appels de procédures à distances** (RPC). La conception actuelle de RPC utilise le protocole de UNIX RPC. Le progiciel informatique à être utilisé dans le projet est ONCRPC, qui est un progiciel expérimental gratuit. Certaines caractéristiques nouvelles ont été ajoutées au paquet pour que le progiciel révisé puisse être intégré au système du NW plus facilement.

C.

La plupart des données représentées par l'Agent Robot sont acquises par les NWs. Les données transmutées entre l'Agent Robot et les NWs seront encapsulées dans les paquets avec quelque format spécial. La transmission et la réception des paquets sont contrôlés par le Manager Communication transparemment. Les paquets aux formats spéciaux est appelé les paquets de communication IR. Avant de parler de la conception structurale de l'Agent Robot, il est nécessaire de décrire la conception des paquets de communication IR.

Les structures originelles des paquets ont été proposées par d'autres étudiants. La proposition a défini quatre types de paquets. Les types ont la même structure d'en-tête, mais les structures de corps sont différentes. Les structures originelles d'en-tête et du corps des les types sont résumées respectivement comme suit:

- La structure originale de l'en-tête est illustrée par la table 4-1. L'en-tête a cinq champs, et occupe sept octets (à 56 bits) en longueur.

- Le type STATUT est utilisé pour indiquer le statut de travail d'un NW. Le corps que ce type possède inclut un champ qui indique le statut d'opération, et trois champs indiquant les trois températures internes du NW.

- Le type DÉPLACEMENT est utilisé pour aviser un NW qu'il faut commencer à se déplacer à la prochaine position. Le corps de ce type comprend au moins cinq champs, comme illustré dans la table 4-3.

- Le type BALAYAGE du STM est utilisé pour aviser un NW qu'il faut commencer le balayage du STM avec les paramètres fournis dans le paquet. Le corps de ce type contient 8 champs, comme illustré dans la table 4-4.

- Le type CONTRÔLEUR PID du STM est utilisé pour dire à un NW qu'il faut régler son contrôleur PID du STM avec les paramètres fournis dans ce paquet. Le corps de ce type comprend sept champs, comme illustré dans la table 4-5.

Les paquets devraient être séparés en deux groupes: un groupe de paquets est envoyé du NWPCS aux robots, qui sont appelés les paquets **down-link**; l'autre groupe de paquets est envoyé des robots au NWPCS, et sont appelé les paquets **up-link**.

Dans la conception originale, la longueur du champ Longueur Du Paquet est seulement d'un octet, ce qui limite la longueur des paquets à un maximum de 256 octets. La longueur du champ Longueur Du Paquet devrait être augmenté à un mot (deux octets) afin que les paquets plus longs qu'un kilo-octet puissent être transmis. Après la longueur du champ Longueur Du Paquet a augmenté, la nouvelle structure d'en-tête du paquet est illustré dans la table 4-7, et le DSP peut contrôler les paquets plus facilement.

Dans l'implémentation actuelle, le champ NWID est utilisé pour spécifier le transceiver ID de le canal de Communication IR, dans un environnement avec multiples cellules de travail. En plus, afin que le temps exigé au recueillement de toutes les données des températures satisfasse aux exigences posées, la conception actuelle attache les données des températures de chaque robot à tous ses paquets up-link.

Le type DONNÉES DE STM est un nouveau type du paquet up-link. Le type DONNÉES DE STM est utilisé pour transmettre (à partir d'un NW) les données des images STM au NWPCS. Le corps des données STM comprend trois mots pour les données de températures, plus une liste de champs des données STM. Chaque champ de donnée STM contient une donnée pixel de 16 bits. La structure du corps de ce paquet est illustré dans la Table 4-8.

Il existe un nouveau type REQUÊTE DES DONNÉES STM aussi, qui est un type de paquet down-link et pour le NWPCS exiger les données de STM. Comme illustré dans la table 4-9, le corps de ce type est seulement un mot de 16 bits. Celui-ci révèle au robot cible le nombre maximum de pixels (mots) qui peut être transporté dans le prochain paquet du type DONNÉES DE STM.

Selon la conception actuelle, chaque paquet du type REQUÊTE DES DONNÉES STM prévoit seulement un segment de données de STM qui lui soit corrélatif. Pour obtenir le prochain segment de données de STM, le robot doit envoyer un nouveau paquet du type REQUÊTE DES DONNÉES STM.

Un nouveau type de paquet, appelé CONTRÔLE DE STM, est une combinaison du type STM SCANOGRAPHE et du type CONTRÔLE DE PID ET DE STM, comme illustré dans la table 4-11. Quand un robot reçoit ce type de paquet, il procède pour commencer le balayage du STM.

D.

L'objectif de construction de ce Manager Communication IR est de créer un mécanisme de transmission des parquets qui soit insensible au contenu des paquets. à l'extérieur de ce Manager Communication IR, l'Agent Robot peut envoyer des paquets down-link dans le manager et lire les réponses up-link du dernier; dans le manager, il est capable d'accomplir tous ces exploits sans prendre soin de la manière dont les paquets sont transmis. Il transmet automatiquement les paquets down-link à leurs robots de cible à la façon d'un paquet à la fois.

Le Manager Communication IR est élaboré pour contrôler seulement une cellule de travail. Le contrôle concomitant de multiples cellules de travail ont besoin de plusieurs managers de communication. La crédibilité du maniement des transmissions des paquets est considérée comme un problème majeur dans la construction du manager de communication. Au gré de la crédibilité, la conception et l'implémentation doivent s'assurer que chaque opération du manager sera strictement régulée. Un autre problème majeur dans la construction de manager de communication est l'efficacité du temps. L'efficacité du temps du manager astreint à ce que les algorithmes du manager soient soigneusement optimisés. En plus, la conception du manager de communication doit être flexible aux changements du matériel communication et doit aussi disposer d'une interface simple pour l'Agent Robot.

Chaque manager de communication possède seulement un tampon entrant pour recevoir tous les paquets provenant des robots qui sont dans les cellules de travail; par contre, ce manager possède beaucoup de tampons sortants pour envoyer les paquets down-link aux robots. Le nombre de tampons sortants est identique au nombre de robots sur la plate-forme de travail. Les tampons sont queues circulaires qui inclut un secteur de mémoire, un pointeur Tête, un pointeur Chevelure, une valeur de capacité de tampons, et un pointeur Fin qui indique la fin dynamique de queue. Ces tampons sont alloués et initialisés quand le manager de communication est chargé dans le noyau. Le premier tampon (le tampon zéro) est utilisé comme le tampon entrant; le reste des tampons est assigné au NWs avec le NWID correspondant respectivement.

La **longueur maximum de paquet** signifie la longueur maximum d'un paquet que le manager de communication peut rencontrer probablement. L'un des usages de la longueur maximum de paquet est de déterminer si le pointeur Chevelure a besoin d'être mis sur le haut de ce tampon. L'inconvénient d'un paquet trop long est que l'usage de l'espace du tampon n'est pas efficace. Donc, la valeur correcte de la longueur maximum de paquet est nécessaire pour équilibrer l'efficacité spatiale de la mémoire et le coût de la transmission.

Le manager de communication de IR est élaboré comme un driver d'appareil caractère qui fonctionne dans le noyau de Linux. Ces types de driver caractère sont traités comme les dossiers de données normaux. Ils peuvent être “ouverts”, “lus”, “écrits” et “fermés” tout

comme un fichier texte. Le manager de communication peut être divisé en onze composants fonctionnels: la fonction Device-write, la fonction Device-read, la routine Package-import, la routine Package-export, les tampons, la routine Timed-package-send, la routine Receiving-interrupt-handler, la portion de matériel contrôle de routine Timed-package-send, la portion de matériel contrôle de routine Receiving-interrupt-handler, la fonction de Device-I/O-control, et les modules fondamentaux pour manipuler le manager dans le noyau. Les relations logiques de ces composants sont illustrées dans la figure 5-3.

La routine Timed-package-send a deux buts: le premier est l'allocation des tronçons d'horaire, et l'autre est la transmission des paquets sous le mode DMA. En général, il consiste en une tâche en tant minuteur et sa fonction de service sous-jacent, une routine expédition avec deux wait_queues et un sélecteur (variable) de robot. La portion de matériel contrôle de routine expédition inclut la routine du maniement de l'interruption de la transmission DMA. La structure de la routine et la logique de contrôle pour les composants sont illustrées dans la figure 5-8. L'allocation des tronçons d'horaire est faite par la tâche en tant minuteur et un wait_queue.

Le manager de communication ne peut pas contrôler et surveiller la procédure de la transmission DMA pendant qu'il reçoit des paquets up-link. La routine Receiving-interrupt-handler est seulement activée après que le PC87108A ait élevé l'interruption end-of-reception.

La routine Package-import, la routine Package-export, la routine timed-package-send et la routine Receiving-interrupt-handler sont indépendants l'un de l'autre. Il y a des sémaphores et des Flags pour coordonner ces processuses correctement. La figure 5-7 illustre la logique de synchronisation de ces quatre routines. Les mécanismes de synchronisations incarnent le rôle des interrupteurs (on/off) dans le processus de coordonner de ces routines. Ils équilibrent la vitesse du traitement des données et la vitesse de transmission IR des données. De plus, ils empêchent le tampon de devenir plein, un phénomène qui peut causer des pertes de données dans des paquets.

E.

Le NWBOS est un programme qui fonctionne dans cadre du NanoWalkers (Veuillez consulter le chapitre trois pour plus de détails). Ce programme peut être considéré comme le niveau la plus basse dans l'architecture générale de logiciel du NWPCS. Cependant, en fait, c'est un système indépendant de NWPCS. Son seul rôle central est le contrôle d'une série de composantes matérielles qui se logent directement dans un NanoWalker;

De plus, il possède une capacité de communication IR. Le DSP contrôle principalement le TIR2000 et le CPLD. C'est le CPLD qui traduit les ordres provenant du DSP et qui ensuite contrôle les opérations du STM. En plus de contrôler celui – ci, le CPLD régit également

- les piézo-'jambes' (une notion spécialisée dans le domaine de la piézo-électricité),
- les LEDs positionnements,
- et les détecteurs de température.

La figure 6-2 démontre l'architecture du logiciel du NWBOS. Il inclut:

- le module principal,
- le module de communication IR,
- le module pour le contrôle des LEDs positionnements,
- le module pour les paramètres (ou variable) public,
- le module pour contrôler de STM et gagner des données,
- le module pour contrôler des déplacements,
- et le module pour l'acquisition des températures.

Le module principal coordonne les opérations d'autres modules. Il accepte les demandes d'opération reçues par le module de communication IR, et traduit ces demandes en ordres pour ensuite les distribuer aux autres modules fonctionnels. Ensuite, il invoque encore le module de communication IR pour renvoyer les données sortantes au NWPCS.

Le module pour le contrôle des LEDs peut être comparé à une routine simple qui met en branle le CPLD afin qu'il puisse déclencher ou juguler le clignotement des LEDs.

Le module de communication IR inclut trois sous-unités fonctionnelles: l'unité pour traiter des paquets, l'unité pour contrôler des transmissions, et l'unité pour contrôler le processus de réception, comme illustré dans figure 6-2. L'unité pour contrôler des transmissions et l'unité pour contrôler le processus de réception sont fournit pour contrôler des opérations matériel du TIR2000. En substance, l'unité pour traiter des paquets est fournit pour analyser des paquets reçus; ensuite, ceux–ci seront métamorphosés en des demandes d'opérations provenant d'autres modules et en des bases de données entrant.

F.

Le test élaboré pour vérifier le bien – fondé de canal de la communication IR consiste (à déterminer) les facteurs suivants :

- la crédibilité du système
- Le tronçon du temps ou d'horaire le plus court possible dans le système
- (en moyen) le temps turnaround du système

Les deux premiers facteurs représentent les deux buts mentionnés ci-dessus. Le troisième facteur consiste à découvrir comment améliorer le canal de communication IR.

Le test de crédibilité est un test important qui devrait être basé sur le grand nombre de transmissions des paquets. Les résultats du test ont montré que tous les paquets up-link ont été correctement reçus pendant le cycle d'allocation. Dès lors, on peut dire que le canal de la communication IR est très crédible.

Le tronçon du temps ou d'horaire le plus court possible est important pour évaluer la prouesse de ce canal. Sa procès peut seulement être estimée avec les expériences. Le test est élabora pour accomplir les mêmes opérations d'envoi et de réception à de nombreuses reprises. Le laps total de temps pris est enregistré. Alors le laps moyen du temps le plus court possible est obtenu en divisant le laps total du temps pris par le nombre total d'opérations d'envoi et de réception. Seulement un robot est impliqué dans le test. Le résultat montre que le laps total du temps pris augmente proportionnellement avec le nombre d'opérations d'envoi et de réception.

Le tronçon du temps ou d'horaire le plus court possible augmente proportionnellement avec la longueur des paquets; et si les paquets sont courts, alors la limitation majeure sur la véritable ampleur de ce canal de la communication IR est déterminée par la vitesse du traitement des programmes dans les PCs du CCMS et du DSP du NW.

Le temps turnaround est aussi un autre facteur important pour évaluer l'efficacité du système. En vue d'obtenir plus précisément les laps de temps totaux pris, l'expérience relatée ici a recours aussi à une panoplie de séquences d'instructions informatiques qui se répètent pour un nombre indéterminé de fois jusqu'à ce que la condition particulière exigée soit atteinte. Les résultats des laps de temps total pris, divisée par le nombre de répétitions des instructions mentionnées ci – dessus, donneront le temps turnaround moyen du système.

Pour la simplicité, le nombre de répétitions des instructions informatiques sera assigné par la valeur de 1,000. Le test utilise de multiples "NanoWalkers virtuels" pour simuler les multiples systèmes de Nanowalkers. Les résultats du test démontrent que le temps turnaround augmente en même temps que les paquets augmentent en longueur.

Cependant, le temps turnaround n'est pas affecté évidemment quand la longueur de paquets est courte. Ceci signifie qu'une majeure portion du temps turnaround est prise par le NWPCS et NWBOS.

Au demeurant, le résultat souligne que le temps turnaround moyen peut être sensible aux algorithmes utilisés par l'OS. De plus, le temps turnaround moyen augmente proportionnellement avec le nombre de robots.

Table of Contents

List of Figures

List of Tables

List of Appendixes

Table of Notations and abbreviations

A/D	Analog to Digital
ASPS	Atomic Scale Positioning System
CCMS	Central Control and Management System
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit of computer
CRC	Cyclic Redundancy Check
DAQ	internal environment Data Acquisition of the cooling chamber
DC	Direct Current
DMA	Direct Memory Access
DSP	Digital Signal Processor
EOF	End of frame
EOT	End of transmission
FIFO	First In First Out buffer
FIR	Fast Infrared
GSPS	Global Scale Positioning System
ID	Identification
I/O	Input / Output
I/V	Current to Voltage
IR	Infrared
IrDA	Infrared Data Association
ISA	Industry Standard Architecture
LED	Light Emitting Diode
NW	NanoWalker
NWBOS	NanoWalker Basic Operation System
NWID	NanoWalker Identification
NWPCS	NanoWalker Platform Control System
OS	Operating System
PID	Proportional, Integral and Derivative
PSD	Position Sensing Device
RAM	Random Access Memory

ROM	Read Only Memory
RX	Reception
RS232	One of the computer serial data communication standard
STM	Scanning Tunneling Microscope
TCP/IP	Transmission Control Protocol / Internet Protocol
TX	Transmission
USB	Universal Serial Bus computer data communication standard

Chapter 1. Introduction

As one of the leading edge in modern technology evolution, nanotechnology [1] provides us tremendous possibilities to design and build devices with molecular and atomic precision, and furthermore to handle atomic size [2] problems. It has been improved significantly during the last decades. Applications of nanotechnology have expended to many areas in industry, including electronics, new material, and biomedicine.

1.1. Introducing the NanoWalker project

The NanoWalker (NW) project [3, 4, 5, 6] was initiated by the BioInstrumentation Laboratory of the Massachusetts Institute of Technology (MIT) under the direction of professor Ian Hunter and professor Sylvain Martel. In May 2002, the project was moved to the NanoRobotique Laboratory of the École Polytechnique de Montréal (EPM), under the direction of Dr. Sylvain Martel. The purpose of the project is to build a machinery system which allows a fleet of miniature robots to detect and manipulate molecular and/or even atomic scale mechanical or biomedical samples simultaneously.

In brief, a NanoWalker is a wireless miniature robot designed for nanometer scale operations with a Scanning Tunneling Microscope (STM) assembly [4, 7, 8] as its actuator. The NanoWalker is able to scan (using STM) the surface of certain samples scattered on a working area. The NW project aims at building a high throughput performance system which allows large number of NanoWalkers to work on a working platform (currently with an area of 0.8m×0.8m[1] [6]) simultaneously. All of the robots[2] in the system are controlled by a Central Control and Management System[3] (CCMS) with wireless communication links. Each robot can move freely on the working platform, and is propelled by a three-piezoelectric-tube assembly [9]. The positioning system of the robots is the combination of the Global Scale Positioning System (GSPS) method and the Atomic Scale Positioning System (ASPS) [5, 10] method.

---

1 The detailed design and the specification values of the components maybe modified from time to time, see more explanations in section 2.1.

2 The current design is aiming at allowing 100 NanoWalkers working simultaneously.

3 The Central Control and Management System is basically a group of computers connected with a network.

In the current stage, since the infrared (IR) transceivers can be made in pretty small size, and do not need very complicated circuit to drive, the infrared communication is used as a wireless communication method for the NanoWalker project. The GSPS also uses IR signal as the medium for delivering positional information.

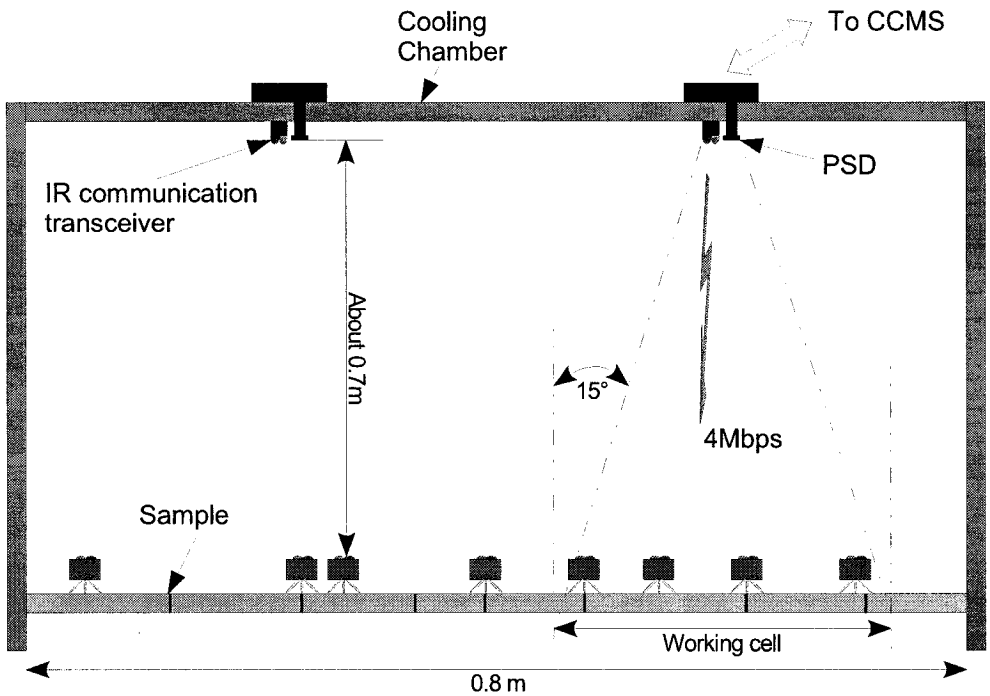

Figure 1-1: The setup of the NanoWalker project.

Figure 1-1 shows the general setup of the NanoWalker system. In this figure, a few Position Sensing Devices (PSD) and IR communication transceivers are mounted above the working platform that has a number of robots working on its surface. The PSDs are used by the CCMS for global scale positioning of the robots, and the IR communication transceivers are used for communication between the CCMS and the robots [4, 6]. Each PSD works together with an IR communication transceiver to control a certain area of the working platform. The area is called a working-cell. The size of a working-cell is determined by the visual angle of the IR communication transceiver. The visual angle, which is specified by the IrDA[4] specification [11], is about fifteen degree. All of these

4 The infrared communication uses the Fast Infrared (FIR) as its data link [12] protocol, which is specified by the IrDA.

hardware components, except the computers of the CCMS, are enclosed in a cooling chamber, which provides cold air to regulate the internal temperature of the robots to a proper range [6].

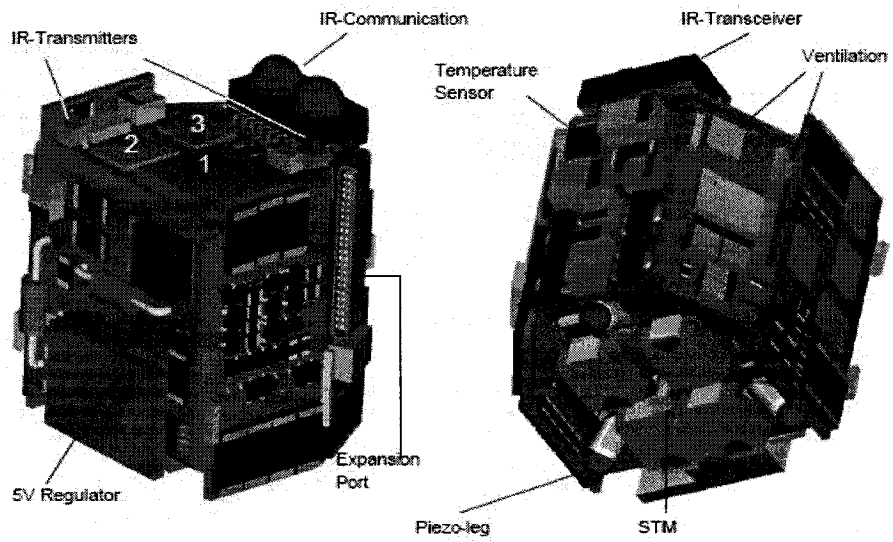

Figure 1-2: The external design of a NanoWalker [6].

Figure 1-2 shows the external view of a current version NanoWalker which has a size of approximately 32×32×30 millimeters (mm) [6]. As shown in the figure, there is an IR communication transceiver mounted on top of each NanoWalker, which is used for IR communication. At the diagonal of the robot, there is a pair of IR positioning LEDs, which give off the positional and oriental information. This positional and oriental information will be received by the PSD(s) and passed to the CCMS. The PSD is a very sensitive device. With the PSD(s) above the working platform and the pair of positioning infrared LEDs on the NanoWalkers, the GSPS is able to provide an optical positioning method with a resolution as fine as tens of micrometers [5].

1.2. The objective

As mentioned above, the project was started in 2001. Quite a few parts of the project, including the controlling software architecture, had already been in progress before the research for this thesis was started in Jan. 2004. The research is concentrated on the software development for the NanoWalker system, and the objective of the research mainly

involves the following aspects:

- Continue the design of the controlling system architecture, and make any necessary modification on the previous design.
- Design and implement the infrared communication manager, and furthermore, ensure the infrared communication channel between the CCMS and the robots to be ready for integration.
- Try to make any possible integration on the finished parts into the controlling system, and find out any problems of their design proposals through a series of testing.

1.3. The overview

Chapter two further clarifies some basic concepts (specifications and designs) about the hardware architecture used in the NanoWalker project that are related to the research for this thesis. Chapter three describes about the concept of layered architecture. Chapter four describes the detailed design of the data packages used in IR communication. Chapter five describes the detailed design and implementation of the infrared communication manager. Chapter six introduces the architecture design of the program that runs inside the NanoWalkers. Chapter seven is the testing analysis of the IR communication channel. Chapter eight gives a general summary of the thesis and talks about the future work of the project.

Chapter 2. Background hardware knowledge

In addition to the introductory description of the NanoWalker project in chapter one, this chapter gives more background information about the hardware architecture of the NanoWalker project which forms the working platform of a nanofactory. Although this thesis concentrates on the software side of the NanoWalker project, some necessary hardware knowledge has to be aware, because the purpose of the software system is to control a big and sophisticated hardware system. Without sufficient knowledge about the hardware architecture of the project, it is difficult to understand the design of the software architecture and its components.

2.1. About the changes of design

The design proposals and specification values presented in this thesis are based on the newest reference materials. Usually after some experiments being finished, some problems in previous designs will be discovered, and solutions or suggestions for those problems will be proposed. Therefore keeping up with the newest ideas is very important. One can notice the changes of designs when following the relevant references.

2.2. The NanoWalker

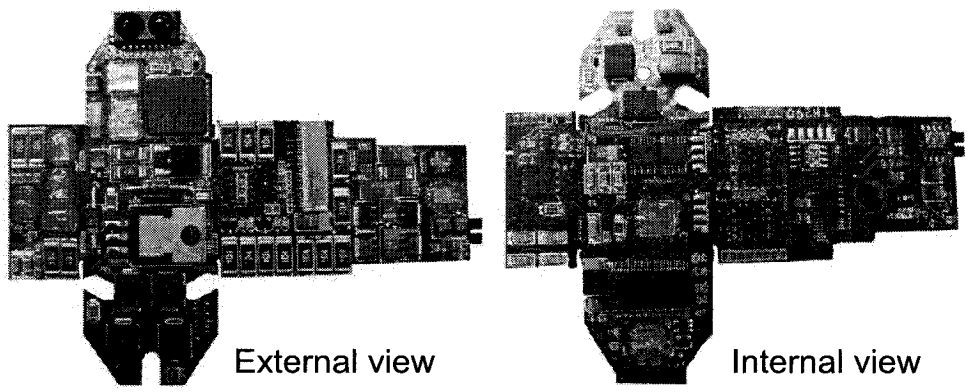

Figure 2-1: The prototype of the flexible circuit board of the NanoWalker [13].

Compared with a highly intelligent robot, the functionality of a NanoWalker is relatively simple. The major difficulty of building the NanoWalker lies on the limitation of

its size. Figure 2-1 shows the prototype of the flexible circuit board of the NanoWalker. The surface of the circuit board is too limited to put a lot of electronic components in. Therefore the NanoWalker cannot have a lot of RAM or a very powerful CPU. The internal space of the robot body is filled with the DC/DC converters as shown in figure 2-2, and there is no place to put any effective power source, not even a bigger battery. Thus one cannot expect that the NanoWalker itself has any artificial intelligent capability, or is able to do complicated jobs independently.

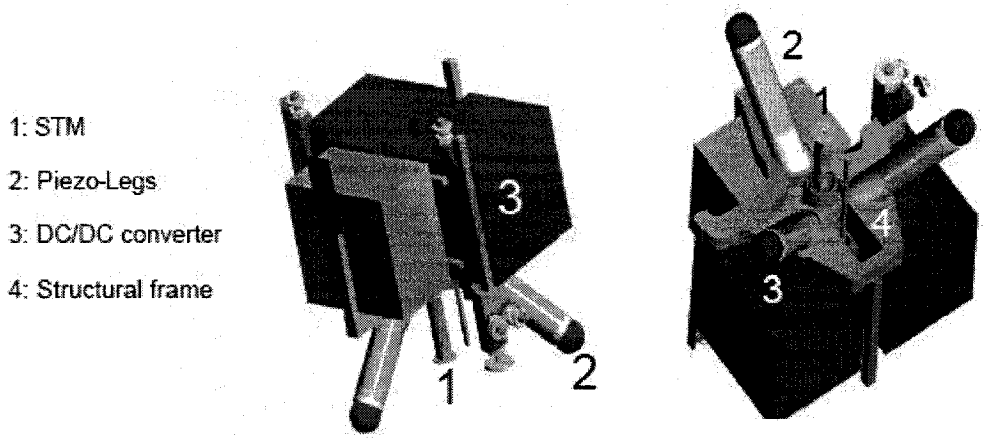

Figure 2-2: The internal structure of the NanoWalker[4]

The only data processing component is the digital signal processor (DSP) TMS320C25-50 [13]. TMS320C25-50 is a 16-bit integer-only processor without floating point calculation, and its memory address space is limited to 128k words[5] (256k bytes) for data and program[6] (RAM and ROM together) and 256 words for I/O addressing [14]. This limitation determines that the DSP can only deal with the basic I/O control of the hardware components while leaving the main portion of the computational work to the CCMS. All of the peripheral components of the NanoWalker, including the STM, the legs, the IR communication controller, etc., are connected with the DSP with a CPLD [15].

The IR communication controller of the NanoWalker is TIR2000, a product of Texas Instruments. The transceiver is HSDL-3600. The TIR2000 has maximum 64-byte FIFOs, multiple IrDA mode support and other features that are particularly suitable for IR communication.

5 In this thesis, a **word** is a 16 bit integer.
6 The present design used 64k words (128k bytes) as data memory.

The internal structure of the NanoWalker, as shown in figure 2-2, has three DC/DC converters which are responsible for providing power to the whole NanoWalker [16]. These DC/DC converters and other electronic components can generate substantial amount of heat [15]. There are three temperature sensors inside the body for detecting the internal temperature [6]. The outputs of these sensors will be converted to three 9-bit integer data, which will occupy three words of data memory. These temperature data will be sent back to the CCMS as a feedback for controlling the cooling chamber.

The locomotion of a NanoWalker is done by a three-piezoelectric-leg assembly as shown on the right-hand-side of figure 2-2. The benefit of using this type of locomotion system is that the robots can have fast movement while still keeping an acceptable high positioning resolution. In the current design, each leg has four electrodes [4, 16] so that it can generate vibrations in two directions. These vibrations in turn cause the whole robot to move in straight lines or rotate. The movement speed of the robot is controlled by the frequency of the vibrations of the legs (frequency modulation). How to coordinate the vibration of the three legs is critical for the movement. Each motion of the legs is controlled by a 12-bit bit sequence. Figure 2-3 shows two examples of the controlling bit sequences. A series of bit sequences defines a series of motions. The motions turn out to be the vibrations of the legs. Since the vibration patterns of the legs are complicated, all of the bit sequences are generated by the CCMS and are transmitted to the robot by IR communication. The DSP is only responsible for converting the bit sequences into electric pulse of each pole.

Linear movement toward leg A direction												
Leg	A				B				C			
Electrode	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3	C4
Bit 0	1	1	1	1	0	0	1	1	0	1	1	0
Bit 1	0	0	0	0	1	1	0	0	1	0	0	1

Figure 2-3: Examples of the 12-bit controlling bit sequences [13].

The actuator of NanoWalker, the STM, as shown in figure 2-2, is also made with a piezoelectric tube with four electric poles. Unlike the legs, the vibration pattern of STM is simpler, and the controlling bit sequence can be generated by the DSP itself as long as the necessary parameters are supplied. These parameters are given by CCMS through the IR communication. The data acquired by the STM, namely the STM data, is the largest portion

of information to be transmitted back to the CCMS through the IR communication. Each pixel of a STM image takes 16 bits [13], which is one word. Since the total RAM of a NanoWalker is only 64K words, to scan a bigger size STM image, the STM image data have to be sent back to the CCMS while the STM is scanning. In that case, the time for the CCMS to acquire a complete STM image data could be quite long. For example, if a STM image has 1k×1k pixels which takes 1M of words, and if the transmission bandwidth of the IR communication channel is 4Mb/s (250kwords), then theoretically it will takes at least 4 seconds to send 1M words of STM data from the NanoWalker to the CCMS. However, if the time slot allocated to the robot is less than 4 second (which is normally the case), then completely sending out the STM image data needs more than one allocation cycles[7].

2.3. The positioning system

Knowing the basic theory of the whole positioning system is to understand the software requirements about the positioning system that are relevant to the design of the robot agent and the IR communication manager. The NanoWalker project uses the GSPS to reach the locations of the target samples with a precision of less than 100μm at first, and then uses the ASPS to further position their STM tips right above the samples with a precision of about 1μm.

The setup of a GSPS assembly, which controls one working-cell, is depicted in figure 2-4. The GSPS consists of a PSD (Sitek PSM2-20), a pair of IR emitting diodes of each robot, a couple of signal converters (an I/V converter and an A/D converter), and a computer for processing the position data of all robots. The recent study indicates that it is also necessary to use the IR communication channel to synchronize the positioning emitters of each robot and the sampling procedure of the PSD. Therefore the IR communication channel can also be considered as a part of the GSPS. The detailed synchronization procedures will be described in chapter six. A feasibility research [5] on the use of the PSD as the position sensor had been done after the NanoWalker project was set up. Thereafter, some other researches had also worked on the GSPS [17], but mainly focused on the software side of controlling the PSD and handle the PSD collected data.

---

7 See chapter seven for the meaning of an allocation cycle.

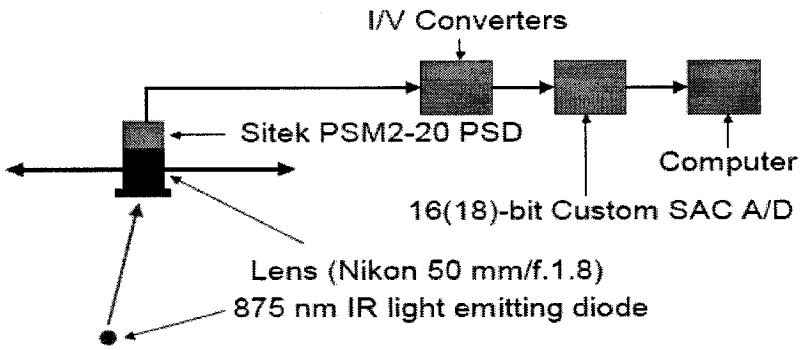

Figure 2-4: The set up of a PSD [5].

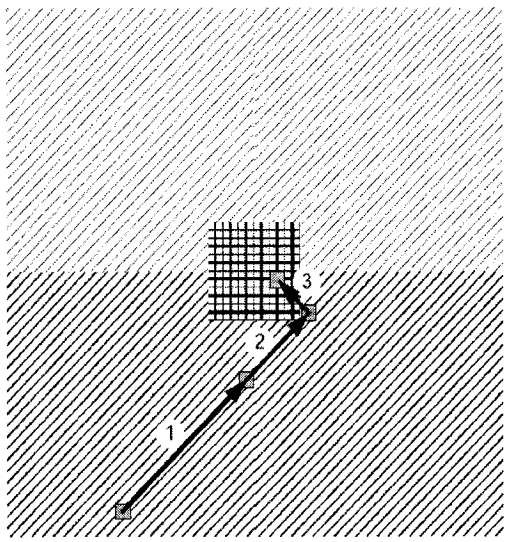

Figure 2-5: The prototype of the grids for ASPS [13].

With regard to the ASPS, a research had been done about deciding the pattern of ASPS using the STM as the positioning device as well [18]. In that research, the positioning method was to make (etch) grids with a special pattern around the target samples. By analyzing the STM image of the grids, the method allowed NanoWalkers to reach the target sample within 3~4 iterations of scanning and moving. Figure 2-5 shows the prototype of the grids and a demonstration of the searching steps of the STM [13]. No more research on the ASPS has been done since then, because the prototype of the STM actuator is not ready yet.

2.4. The IR communication controllers

At the current design and testing stage, an IR communication assembly in the CCMS side consists of an ACT-IR 2000BL ISA adapter card and an ACTisys-IR transceiver. The external view of the card with the transceiver head is as shown in figure 2-6 below. The ACT-IR2000BL ISA adapter card is equipped with an IR communication controller, PC87108A, which is a product of National semiconductor. The reason for this card being adopted in this project was because it was the only suitable product in the market at the moment (year 2002) of developing the prototype of the infrared communication device driver (in Linux). The card now is plugged into an Intel Pentium III PC that has two ISA slots and 320M bytes of RAM. This IR communication controller assembly is for testing one working-cell only.

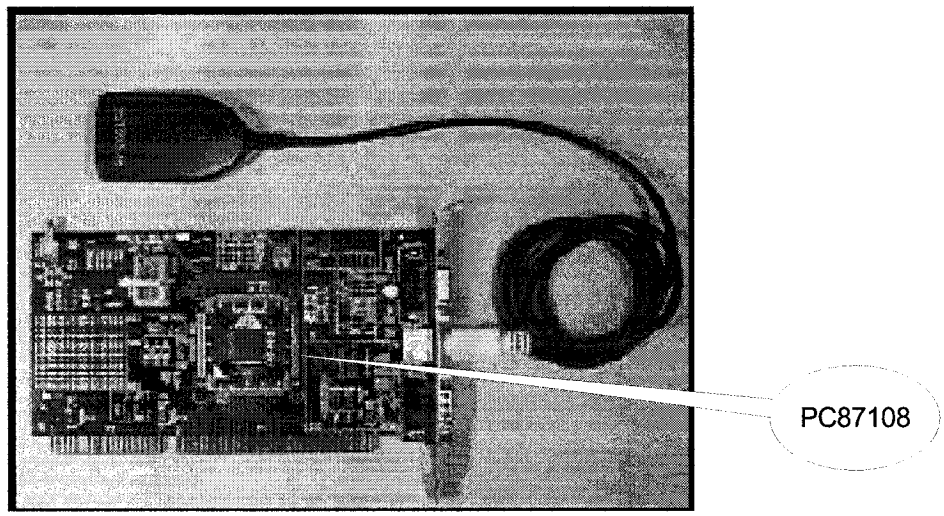

Figure 2-6: The outlook of the ACT-IR200BL IR controller card and the transceiver head.

The PC87108A IR controller can work in multiple IR communication modes[8]. The current design of the IR communication only works in Fast Infrared (FIR) mode with a bandwidth of 4Mb/s. The controller has a 32-level infrared transmission FIFO (**tx-FIFO**) and a 32-level infrared reception FIFO (**rx-FIFO**). It is also equiped with a 8-level DMA status FIFO (**st-FIFO**) which is for recording the status of DMA transmissions. Whenever a DMA transmission (either from PC to the IR controller or vice versa) has been finished, an 8-bit data which describes the status of the DMA transmission will be stored into the **st-FIFO**. The controller can generate an interrupt signal if the level of the **st-FIFO** grows to two or

8 In this thesis, any thing about the PC87108A controller can be referred to the its user's manual [24].

four and the st-FIFO will be reset thereafter.

2.5. The power floor and cooling chamber

The working area of the NanoWalkers is a power floor mounted on a damped base table. Since the NanoWalker has no internal battery, the power floor is responsible for providing the energy [13]. The power floor supply does not need any software.

As mentioned above, the large amount of heat generated by the robots is dissipated into the cooled air in the cooling chamber. The flow of the air is controlled by a computer. The hardware and software design to control the cooling chamber is now a separated project undertaken by another research, and is not an issue concerned with this thesis.

2.6. The computers in the CCMS

The CCMS consists of a set of PC computers. Each of them is responsible for a different aspect of the controlling tasks. The current implementation is planning to use four PC computers to control robots within a single working-cell: one is responsible for running the NanoOS[9], one for controlling the cooling chamber, one for controlling the PSD and GSPS, and one for controlling the IR communication. These computers run on different operating systems. Figure 2-7 shows the network connections in the NanoWalker project[10]. In figure 2-7, the arrows are representing the data flow among the computers.

The machine used for controlling GSPS is connected with the machine used for controlling IR communication. These two machines are the ones that are relevant to the research of this thesis. They form the platform on which the robot agent including the position manager and the IR communication manager run. See the next chapter for more descriptions about the robot agent, position manager, IR manager and the inter-computer communication.

There are a few reasons to use a set of computers instead of one:

---

9 See next chapter for the description about NanoOS.

10 The external user application(s) may run on other computer(s) that are generally not considered as part of the hardware components in the NanoWalker system.

- To increase the system compatibility. Different drivers need different operating systems. The device driver of the PC87108A is written for running in Linux, but other device drivers are written for running in Windows. In such case, it is easier to use different computers to run different operating systems.
- To enhance the capacity and performance. Some modules in the NWPCS such as the IR manager and the PSD manager are time critical. Using a single processor computer is hard to guarantee that all deadlines can be met.
- To share the burden of computation. The whole NWPCS has that many modules with many tasks run together, and requires substantial amounts of computing resources including RAM, CPU time, device I/O, etc. Distributing the modules into different machines can share the computational burden.

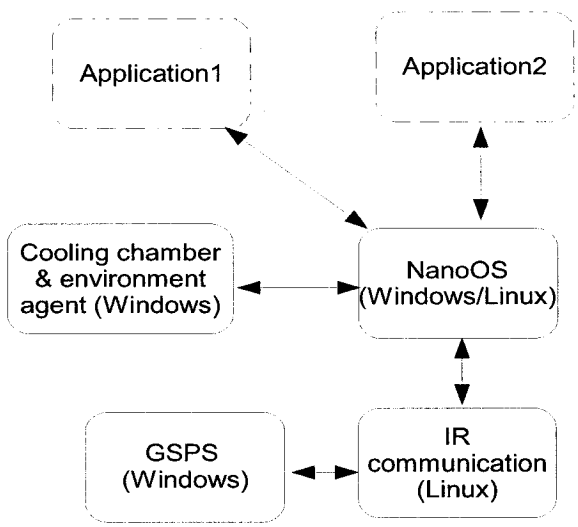

Figure 2-7: The network connection relationship of the computers in the NanoWalker project.

The computers will be linked together with Ethernet. Compared with USB, RS232, IR or other connection methods, the benefit of using Ethernet is that it is easier to set up and there are many supporting software with high stability in hand. However, Ethernet has a drawback that because of the existence of collisions, the time for a package to be successfully received is not guaranteed. Fortunately, since there is not so much data traffic in the network at the current stage, the delays caused by collisions can be ignored. If really necessary, some connections can be replaced by USB or other connecting methods in the future.

2.7. Summary

This chapter briefly introduced the hardware architecture of the NanoWalker system. It provides some preliminary knowledge about the NanoWalker system, including the structure of the NanoWalker, how the NanoWalker moves, how the STM works, how the GSPS and ASPS work, the IR communication assembly, the computers used in CCMS, the cooling chamber, and the power floor. In addition to the above descriptions, one can refer to the NanoWalker project description portion of other documentations [13, 15] for more details.

Chapter 3. The layered architecture of NWPCS

The software system of the CCMS, namely the NanoWalker Platform Control System (NWPCS)[11], is responsible for controlling the whole NanoWalker system. It does not include the basic I/O controlling program which runs inside the DSP of the NanoWalker. As the control software of a high throughput automatic system, the NWPCS should be able to have real-time response to the changes of whole system, and the output of its responses must be reliable. More importantly at the current stage, because most components of the hardware architecture are still under planning, the software architecture should be relatively independent to the hardware components and flexible to the changes of specifications of the hardware components.

As early as the project was still in MIT, there was already a preliminary program developed by students in MIT for doing some basic controls. The program was written in one piece, running in Windows only. The lack of documentation and extensibility made it completely not reusable. After the project was moved to École Polytechnique, this preliminary program was abandoned. The work on developing the architecture of the NWPCS restarted by students [13] in Polytechnique, and a framework of a new version of the NWPCS was proposed. The proposal included the considerations about the global structure, positioning, IR communication, STM, etc [13]. One aspect of the research for this thesis is to continue the work on designing the NanoWalker system architecture based on this proposal and give more designing details to the whole framework. The work also includes following duties:

1. whenever the designs of subsystems are finished, and the related hardware components are available, try to implement the designs to check the effects;
2. based on the above-mentioned implementation results, propose the necessary improvements on the designs.

The structure of NWPCS was divided into multiple layers. The rest of this chapter will go through the design of the layered architecture, including the software structure inside

11 The original name of NWPCS is called the Platform Control Software.

NanoWalker which was not mentioned in the previous documentations. In each part, the original design (by other students), the new considerations and the modifications or new features will be described accordingly.

3.1. The advantages of layered architecture

The design of the NWPCS requires that its architecture be as flexible to the changes of the hardware components as possible. A layered architecture in the NWPCS has the following advantages:

1. Compared with a single-layered architecture, the logical relationships among the modules inside a multi-layered architecture can be more easily understood and the problems encountered during implementations can be more easily isolated.

2. The interface between every two layers can be simplified and standardized as much as possible so that each layer is relatively independent. Currently, many hardware components are not ready yet, and their actual characteristics are not clearly known. The independence of the layers allows some upper layer components to be built before those hardware components are available. On the other hand, since the design of the detailed architecture is not mature, the independence of layers also allows that the internal modification of any layer will not affect other layers too much.

3. Some people in the NanoWalker group are now beginning to think about the next generation of the NanoWalker system. For the new requirements of the future specifications, the layered architecture allow people to identify which software components need to be replaced, which need to be modified, and what should be added.

Of course, despite that some upper layer components can be built in advance, the nature of being a controlling program determines that the features and characteristics of most software components have to follow the features and characteristics of the hardware components. That is why in general, the development of the NWPCS is from the bottom up.

3.2. The original architecture

The original architecture proposed by other students had six layers. Starting from the lowest, the layers are: hardware layer, driver layer, manager layer, agent layer, NanoOS layer and the application layer, as shown in figure 3-1. The functionality of each layer is briefly described as the following:

1. The hardware layer includes all hardware components of the controlling platform (excluding the NanoWalker robots).
2. The driver layer containing the device drivers which provide hardware accessing functions for the upper layers.
3. The components in the manager layer, namely the managers, are responsible for converting the controlling data from the format used by the Agent layer to the raw format used by the hardware components, and vice versa.
4. The agent layer consists of agents, which can abstractly represent and coordinate the hardware components (e.g. the NanoWalker, cooling chamber, etc) using the data collected from the manager layer.
5. The NanoOS layer contains one complicated unit, the NanoOS, which coordinates the operations of the agents (intelligently) in the lower layer, and provides a uniform interface for the user applications in the application layer.
6. The application layer includes any user application which utilizes the platform of NanoWalker system.

According to figure 3-1, the agent layer includes an Environment Agent and a Robot Agent. The Environment Agent manages a DAQ (an assembly containing some sensors in the cooling chamber) manager and a Room (the fan and valve controller of the cooling chamber) manager. The Robot Agent handles a communication manager and a Position manager. All managers are in turn responsible for managing their related drivers and hardware. Components within every layer are independent to each other, and if they want to share some information, the shared information has to be passed through their upper layer.

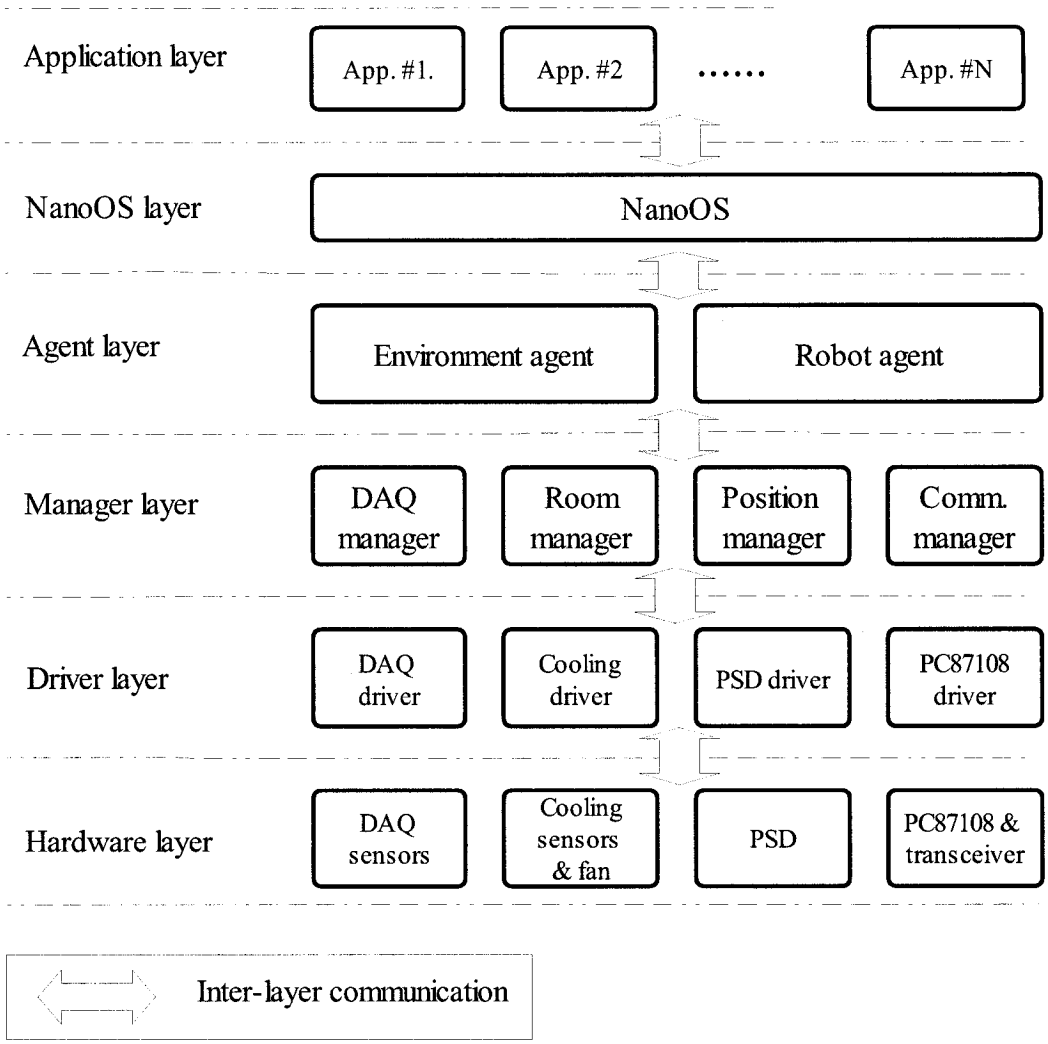

Figure 3-1: The original layer architecture proposed by other students.

As proposed in the original proposal, the across layer data will be delivered by means of inter-layer communications, which in fact are the TCP/IP network communication links among computers.

3.3. Reconsidering the original architecture

The original design provided a big picture of the architecture which had the advantages as mentioned at the beginning of this chapter. Nevertheless, after careful consideration, especially by reviewing the problems met by other people in their researches, some conceptual problems of that design need to be sorted out before digging into more detailed designs of the architecture.

3.3.1. Substituting the hardware layer with the NWBOS layer

The hardware layer is not necessary, simply because it is not in the scope of software. On the other hand, if one really wants to keep a six-layered architecture, the program which runs inside the NanoWalker body can be viewed as the lowest layer instead. Of course, the internal program of NanoWalker runs independently in each robot and plays a very different role from the components running inside the NWPCS and thus should not be considered as part of the NWPCS. It is better to call this program the NanoWalker Basic Operating System (NWBOS), as a counterpart of the NWPCS. The architecture of the NWBOS will be described in chapter six.

3.3.2. Clarifying the role of the driver layer

In the device layer, device driver should be built up to the point that no hardware relevant operations need to be cared by the managers. In the original proposal, the functionality of the device layer has not been clearly pointed out. When developing new device drivers, the developer may feel a difficulty of identifying what features belong to the drivers and what features belong to their managers. In fact the function of device drivers is to isolate all hardware related operations from other software components in the NWPCS. Thus the device drivers not only provide hardware accessing paths (functions) to the upper layer, but also set up the registers and I/O parameters in the hardware components according to the usages.

3.3.3. Modifying the role of the manager layer

Instead of converting data between the agent layer and the driver layer, the role of the manager layer is to coordinate the device driver and ensure that the agents have credible physical data about the NanoWalker system and the samples. Looking at the communication manager, the major works of the communication manager are time scheduling and buffer management, which will be mentioned in chapter five. The major task of the position manager is to determine the most possible coordinates of the robots from a large amount of sampling data.

In addition, the NanoWalker project aims at controlling as many robots as possible. In

such case, the system will need multiple PSDs and IR communication transceivers in the future[12]. It can be expected that there will be multiple communication managers and multiple position managers in the layer (rather than only one communication manager and one position manager, as shown in figure 3-1). Each of them may need to manager one or more device drivers[13].

3.3.4. Replacing the inter-layer communication concept

The inter-layer communication concept should be replaced with the inter-computer concept. The application interface between two layers needs to be as simple as possible but does not need to be a physical separation between two computers. The original proposal in fact assumes that any two layers could not coexist in one computer. Such an assumption about the inter-layer communication seems not very precise. The layers are simply conceptual entities which help grouping modules having similar features together and making them relatively independent from others. Obviously, one should be able to understand that putting a device driver and its managers in two computers is not proper in most cases, because the workload of a device driver is usually too light to be worth occupying a whole computer. On the other hand, if putting a component with very heavy workload, for example the whole Robot agent, into a single computer, the component may be too slow to work properly in real-time, in such case, it is better to distribute the overloading component to different computers. In other word, it is necessary to distribute the resource rationally.

3.4. The revised layered architecture

Based on the new considerations mentioned above, a revised layered architecture is proposed here as shown in figure 3-2. In this architecture, the lowest layer is now the NWBOS layer, which means that the NWBOS program resides in every NanoWalker. The communication links between the NWBOS layer and the device layer can be considered as a special type of inter-computer communication links – the infrared communication links. Other inter-layer data exchanges are done by individual function calls. The manager layer

12 Each pair of a PSD and a IR transceiver controls a single working-cell, as mentioned in chapter one.
13 Currently, each manager is designed for managing only one driver.

now has multiple position managers and communication managers. Each of them manages one corresponding device driver. In addition, the changes about the roles of the device layer and the manager layer can be seen in the design of the IR communication manager as will be mentioned in chapter five.

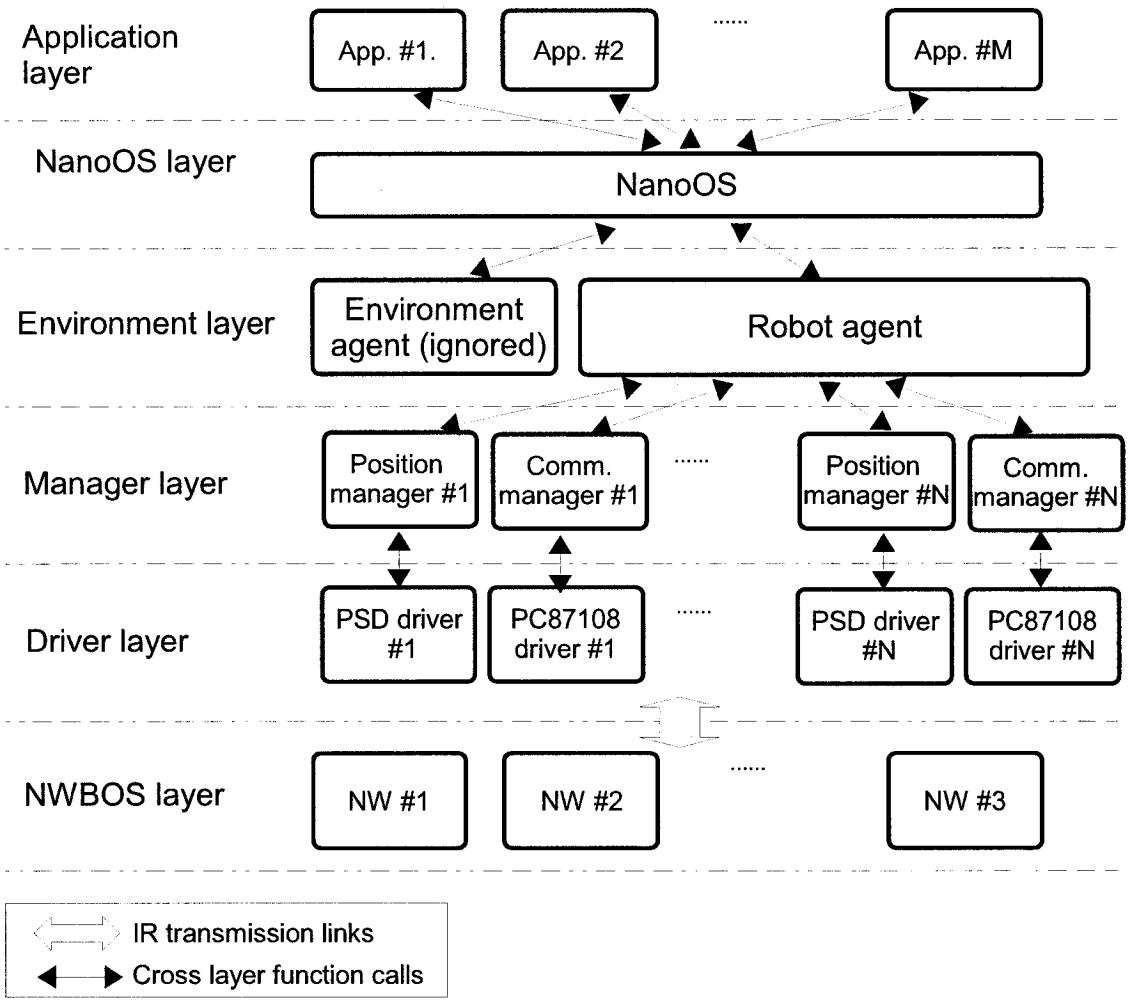

Figure 3-2: The revised layered architecture based on new considerations.

Since everything of the environment agent is basically related to the cooling chamber, and the cooling chamber part is now studied by another student, this thesis will not give more detailed description about this agent.

3.5. Inter-computer communication

The inter-computer communications in the NanoWalker system can be achieved by using remote procedure calls (RPC) [19]. Simply speaking, RPC is a technology that allows one function in one computer platform to call other functions in different computer platform just as all of them are within one platform. Figure 3-5 demonstrates how RPC works.

In figure 3-3, function A is supposed to call function B, but function B is in another computer. What RPC does is that it creates a **server stub** for function B and a **client stub** for function A.

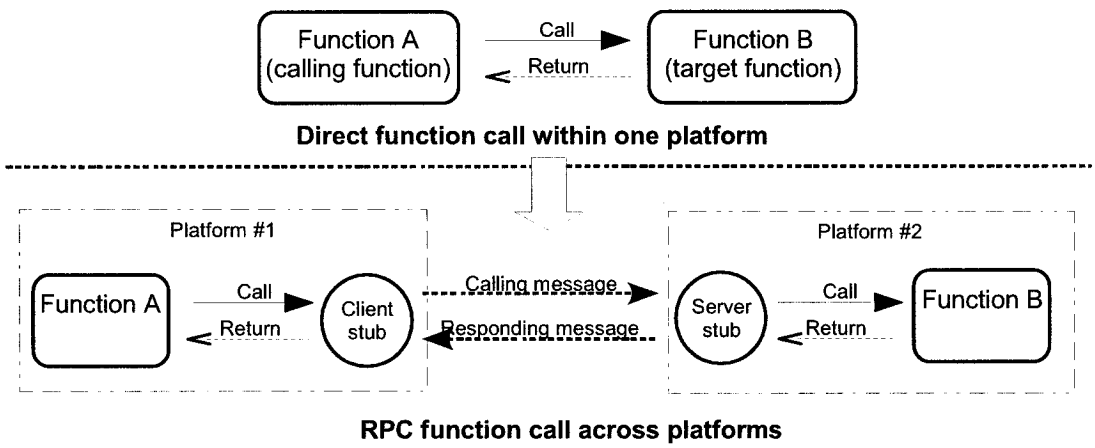

Figure 3-3: The remote procedure calling scheme for function in different platform.

A server stub is a network module specially created for a target function (function B). It listens to a port of the network for calling messages from client stub(s). A calling message to the server stub encloses the identification of the server stub, the identification of the client stub, and the parameter data of the target function. When the server stub receives a calling message (from the network), it calls the target function with the parameter data. When the target function finishes, it returns the result to the server stub. Then server stub then passes the result as a responding message back to the client stub (over the network).

A client stub is also a network module that is created specially for a calling function (function A). To start, the calling function (function A) calls the client stub with its parameter data, and the client stub passes a calling message to the server stub (over the

network). After the client stub receives the responding message from the server stub, it returns the result (from the target function) to the calling function.

The current design uses the UNIX RPC protocol. The benefits of using this protocol are that it is simple to use, and more importantly, there are software packages that allow this protocol to be used in Windows platform. The software package used in the project is ONCRPC, which is a free experimental package. However, a big problem appeared when the package was introduced – the version of ONCRPC adopted here was not actually executable. Causes of this problem included missing declarations of variables and improper function algorithms. A pretty long time had been taken to fix the problem. Some new features were added to the package in the meantime so that the revised package can be integrated to NanoWalker system easier. Now the package is working well and is ready to be used in the project.

3.6. Conclusion and Discussion

This chapter describes the design of the layered architecture. Since the NanoWalker project was moved to École Polytechnique, the layered architecture has become the base framework of the whole design. The order of the developing of the components and all of the logical relationships among the components are based on the layered architecture. In fact, as the process of the whole architecture design getting more and more detailed, the layered architecture is more and more helpful in identifying the features and the importance of the components.

Without considering the application layer, the revised layered architecture is a hierarchical structure, with the NanoOS as the root. Each component has no multiple dependencies on its upper layer components. Therefore such kind of architecture can maximize the independence of every component.

It is worth emphasizing here: as shown in figure 3-2, it is possible that all data exchanged between the driver layer and the NWBOS layer go through a 4Mb/s IR communication channel within one working-cell[14]. Since the computers of the NWPCS can

14 In such case, all robots are within the working-cell.

process data a lot faster than 4Mb/s, it is expected that the 4Mb/s bandwidth limit could become a contingent **bottleneck** of the IR communication channel. At the current stage, there is no better wireless communication method that can substitute the infrared communication method. The system has to stay with this bandwidth limit. How to fully utilize the bandwidth is one of the major concerns in designing the IR communication manager.

The design of the NanoOS, the robot agent and the position manager are not going to be described in detail in this thesis. Appendix B shows a prototype of the position manager for testing the resolution of the GSPS. The design of the robot agent is still undergoing. The relationships of the robot agent with the components in other layers are outlined in appendix A, and the description of how the position manager works is introduced in appendix B. The design of the IR manager will be described fully in the chapter five.

Chapter 4. The IR communication packages

Most of the data represented by the robot agent is from the NanoWalkers. The data exchanged between the robot agent and the NanoWalkers will be encapsulated into packages with special format. The transmission and reception of the packages are handled transparently by the communication manager. This kind of special format packages are called the IR communication packages. Before talking about the structural design of the robot agent, it is necessary to describe the design of the IR communication packages first. The design of the communication manager will be described in detail in the next chapter.

The original structures of the packages were proposed by other students [13]. However, as the design and implementation of the IR communication channel (including package handling portion in the NWBOS) was going on, some problems about the original design were encountered. This chapter is going to explain and summarize original design of the packages at first, and then gives new considerations and solutions to those problems.

4.1. The original package design

The original design of the package structure defined four types of packages. Each type has the same structure of header, but has different structure of body. The structure of the header and the body of each package type are summarized respectively as the following:

4.1.1. The package header

The structure of the original header is as shown in table 4-1. The header has five fields, and occupies seven bytes (56 bits) in length.

- Field 1 is the **NanoWalker Identification** (**NWID**) number. It takes one byte (8 bits). Every robot needs to be assigned a unique identification number. ID number zero is reserved for broadcasting. In other words, the design of the package supports up to 255 robots maximum.
- Field 2 is the total **Package Length**. It takes one bytes (8 bits). The length is expressed in number of bytes, including the length of the header itself. In other

words, the maximum length of the package is 256 bytes.

- Field 3 is the **Reference Package Identification** number. It takes two bytes (16 bits). It tells the receiver of the package to which package (sent by the receiver) it is responding.
- Field 4 is the sender **Local Package Identification** number. It takes two bytes. The range of the package ID is from zero to 2^{16}-1 (65535), which is basically enough for all package IDs (being handled) to be unique at runtime.
- Field 5 is the **Message Type**. It takes one byte. It is important for the receiver of the package to determine how to deal with the body of the package.

Table 4-1: The structure of an IR communication package header.

Field 1	Field 2	Field 3	Field 4	Field 5
NWID	package length	reference package ID	local package ID	massage type
8 bits	8 bits	16 bits	16 bits	8 bits

4.1.2. The STATUS type package

The **STATUS** type package is used for indicating the working status of a NanoWalker. The body of the STATUS type package has five fields and takes only seven bytes. It includes one byte for the operating status and three words (six bytes) to represent the three internal temperatures of the NanoWalker. Each temperature field actually uses 9 bits only. It is possible to compress all of the temperature fields into two words, but to be handled fast by the DSP and to prepare for any possible expansion in the future, it is better to keep them in one word each. The structure of the package body is as shown in table 4-2.

Table 4-2: The body structure of the STATUS type.

Field 6	Field 7	Field 8	Field 9
Status	Temperature 1	Temperature 2	Temperature 3
8 bits	16 bits (9 bits used)	16 bits (9 bits used)	16 bits (9 bits used)

The status field has three bits representing the operating states of a NanoWalker, and for

each state, the other five bits can further indicate a 32-level sub-state. The arrangement of the bits[15] is as shown in figure 4-1.

ready	STM	SDR	5-bit sub-state

Figure 4-1: The bit arrangement of the STATUS field.

4.1.3. The DISPLACEMENT type package

The **DISPLACEMENT** type package is used for instructing a NanoWalker to displace to the next position. As mentioned in chapter two, NanoWalkers lack of the computability to generate vibration sequences of the legs by themselves. Therefore instead of giving the coordinates of the destination, the DISPLACEMENT package directly gives a robot the vibration sequences of all legs. Specifically speaking, the package provides a series of moving steps, and in each step a 12-bit sequence indicates the voltage value of every electrode of every leg [13]. In the original design, each bit sequence occupies one word.

The body of the DISPLACEMENT package has at least five fields including one 8-bit reserved field, one 8-bit field indicating the number of sequences in the package, one 8-bit field indication the frequency used by the legs, one 16-bit field indicating the number of repetitions of the step series, and some 16-bit sequence fields. The number of sequences can be as many as 256. The structure of the package body is as shown in table 4-3.

Table 4-3: The body structure of the DISPLACEMENT type.

Field 6	**Field 7**	**Field 8**	**Field 9**	**Field 10**	**Field 11**	**.....**
Reserved	# of Seq.	Frequency	# of repetitions	Bit sequence #1	Bit sequence #2	.....
8 bits	8 bits	8 bits	16 bits	16 bits (12 bits used)	16 bits (12 bits used)	.....

4.1.4. The STM SCANNING type package

The **STM SCANNING** type package is used for instructing a NanoWalker to start STM

15 STM here means STM scanning, and SDR here means STM Data Ready.

scanning with the parameters provided in the package. The detailed design of the STM is not in the scope of this thesis. One can refer this to other documentations for more details [13].

The body of the STM SCANNING type has 8 fields (15 bytes) including one 8-bit reserved field as well as seven 16-bit fields indicating the fast x and y increments, slow x and y increments, fast x and y update period, and number of fast axis updates respectively. The detailed meanings of the fields are ignored here, and the structure of the package body is as shown in table 4-4.

Table 4-4: The body structure of the STM SCANNING type.

Field 6	**Field 7**	**Field 8**	**Field 9**	**Field 10**	**Field 11**	**Field 12**	**Field 13**
Reserved	Fast Vx increment	Fast Vy increment	Slow Vx increment	Slow Vy increment	Fast Vx update	Slow Vy update	# Fast Axis updates
8 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits

4.1.5. STM PID CONTROL type package

The **STM PID CONTROL** type package is used for telling a NanoWalker to set its STM PID controller with the parameters provided in this package[16].

The body of the STM PID CONTROL type consists of seven fields (13 bytes), including one 8-bit reserved field, five 16-bit fields of PID terms, and one 16-bit field of the logarithm value of STM set point. The structure of the package body is as shown in table 4-5.

Table 4-5: The body structure of the STM PID CONTROL type.

Field 6	**Field 7**	**Field 8**	**Field 9**	**Field 10**	**Field 11**	**Field 12**
Reserved	PID term #1	PID term #2	PID term #3	PID term #4	PID term #5	ln(setpoint)
8 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits

16 The detailed explanation of how the PID controller works is ignored.

4.2. Reconsidering and revising the original design

The original design provides the first version of the IR communication package formats. Whereas during the development of the IR communication channel, many problems of this design were discovered. As a result, some of the package formats need to be modified. The following sections discuss the new considerations about these problems as well as their solutions.

4.2.1. Dividing the packages into two groups

The original proposal about the packages did not point out the relationships among the packages. In fact the original package types were not enough to represent the logical relationships among the data to be transmitted through the IR communication. Some new package types need to be added.

From the transmission direction point of view, the packages should be separated into two groups: one group of packages is sent from the NWPCS to the robots, which are called the **down-link** packages; the other group of packages are sent from the opposite direction, and are called the **up-link** packages. Classifying the packages into these groups helps clarifying how the packages are related to each other, and what operations need to be done before and after the packages are transmitted (by the NWPCS and/or by the NWBOS).

4.2.2. Expanding the Package Length field

In the original design, the **Package Length** field is only one-byte long, which limits the length of the packages to 256 bytes maximum. As will be seen in chapter seven, when transmitting a short package by the IR communication channel, the time used for controlling the sending and receiving procedures is quite significant compared with the total transmission time. Such a controlling time is not going to change with the variation of package length. In other words, the longer a package is, the less percentage of the overhead for controlling the transmission procedure is used.

Since the current implementation of the IR communication channel has been able to transmit more than 1k bytes reliably (as confirmed by the reliability experiment in chapter

seven), the **Package Length** field should be expanded to one word so that packages longer than one kilobyte can be transmitted. In this regard, the **reserved** field in most types of packages, as depicted in section 4.1, can be omitted, saving the overhead of transmitting and handling the useless data. After expanding the **Package Length** field, the header structure seems to be as in table 4-6.

Table 4-6: The header structure after expanding the Package Length field.

Field 1	Field 2	Field 3	Field 4	Field 5
NWID	package length	reference package ID	local package ID	massage type
8 bits	16 bits	16 bits	16 bits	8 bits

4.2.3. Revising header structure

The structure in table 4-6 has a problem while being handled by the DSP. Because the DSP can only handle 16-bit data, all of the 16-bit fields have to be broken into two parts whenever the header is read/written by the DSP. Therefore it is better to put the **Message Type** field in front of the **Package Length** field so that the positions of the **Package Length** field and the two **package ID** fields are consistent with the addressing scheme of the DSP. In fact as the expansion of the **Package Length** field, the whole package header is now in even number of bytes, which matches the addressing scheme of the DSP. The new structure of the package header is as shown is table 4-7.

Table 4-7: The revised structure of the package header.

Field 1	Field 2	Field 3	Field 4	Field 5
NWID	massage type	package length	reference package ID	local package ID
8 bits	8 bits	16 bits	16 bits	16 bits

4.2.4. The NWID field in up-link packages

The original proposal of the package structure did not specify what should be filled in

the **NWID** field of the up-link packages. In the previous implementation of the NWBOS, students had the field filled with the sending robot's ID number so that the NWPCS can easily distinguish the package senders. In fact, this is not necessary because the NWPCS can uniquely identify the sender of an up-link package by checking the **Reference Package ID** in the package, provided that the NWPCS has registered the information of the outgoing packages (to somewhere) before sending them out. Figure 4-2 shows how this identification works.

In figure 4-2, before sending a down-link package to Robot01, RobotAgent registers the **Local Package ID** and the **NWID** to a memory location. Then when RobotAgent receives the responding up-link package, it can use the **Reference Package ID** and the registered information in the memory location, to find out that the (up-link) package is sent by Robot01.

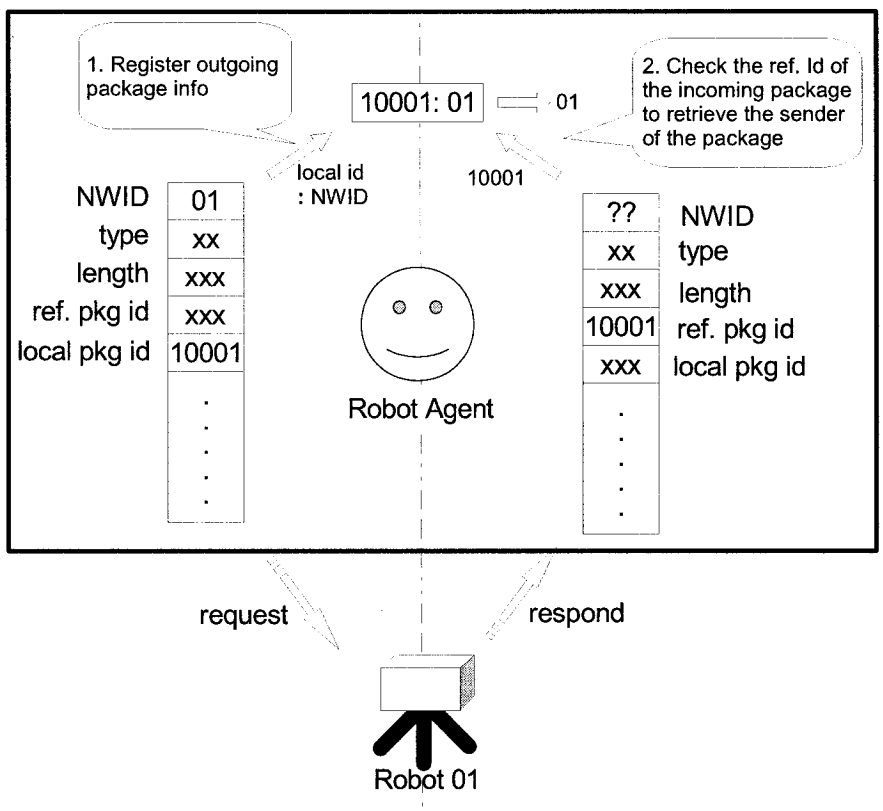

Figure 4-2: Retrieving the sender of an up-link package with its Reference Package ID.

In other words, the **NWID** field of the up-link package is not necessary, and it can be reserved for other purposes. In the current implementation, this field is used for specifying the transceiver ID of the IR communication channels in a multi-working-cell environment. Because each working-cell is circular, there are always common areas that belong to two or more working-cells so that the working platform can be fully covered (by the working-cells). A robot may be right in a common area when it is sending an up-link package. In that case, it may need to specify which transceiver receives the package.

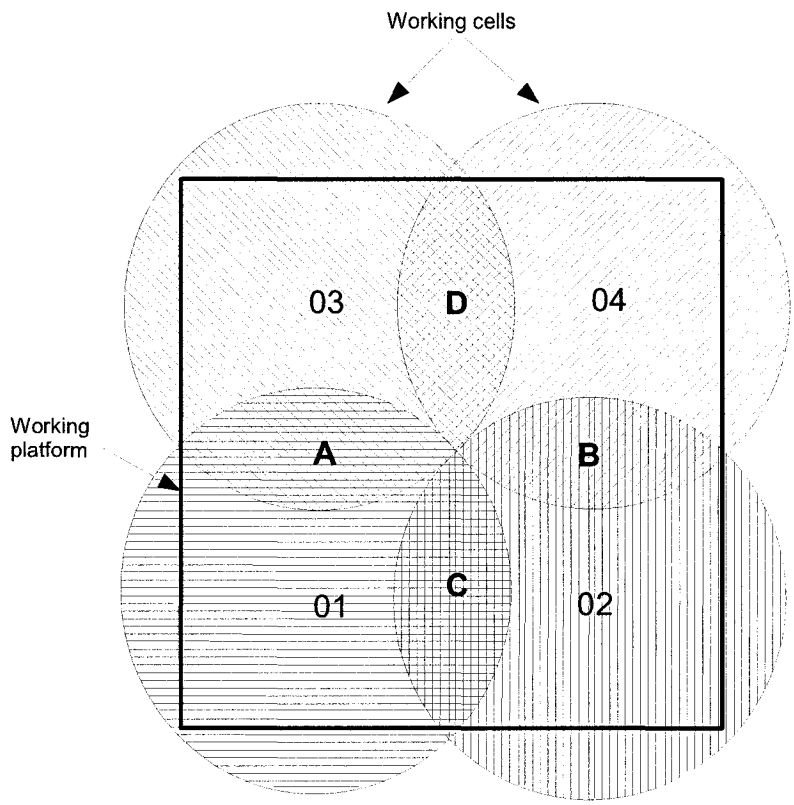

Figure 4-3: The common areas between every two working-cells.

Figure 4-3 shows an example of the common areas. In figure 4-3, A, B, C and D are the common areas; and 01, 02, 03 and 04 are the transceiver IDs of the working-cells. When a robot is inside a common area A, the packages which it sends can be received by all of the related transceivers (i.e. transceiver 01 and 03). In such case, the **NWID** field of a package can be used for specifying transceiver 01 to handle the package.

4.2.5. Associating temperature fields to up-link packages

In current design, the cooling chamber controlling system requires the instant internal temperatures of all robots every one to four seconds. To ensure that the time for collecting all temperature data meets the requirement, the current design attaches the temperature data of each robot to all of its up-link packages. The three 16-bit temperature fields immediately follow the package headers. The result is that the length of an up-link package is at least 14.

4.2.6. The packages for transmitting STM data

The sample image data obtained by the STM is called the **STM data** in this thesis, and the whole block of image data is called the **STM data array**. A larger STM data array has to be broken into many segments and carried by many STM data packages. As mentioned in chapter two, each pixel of the STM data takes one word (16 bits). The size of the STM image could be as large as 1024×1024 pixels, as one of the specification of the NanoWalker project [13]. In other words, the amount of data to be transmitted to NWPCS can be as large as over two megabytes. The limitation of a FIR frame is two kilobytes, as specified by IrDA. Therefore it is impossible to put the whole STM data array in one package.

The STM DATA type is a new up-link package type. The STM DATA package is used for conveying (from a NanoWalker) the STM image data of the target samples back to the NWPCS. The body of the STM data has three words of temperature data, plus a list of STM data fields. Each of the STM data field contains a 16-bit pixel data. The detailed meaning of the pixel data is ignored here, and the structure of the package body is as shown in table 4-8.

Table 4-8: The body structure of the STM DATA type.

Field 6	**Field 7**	**Field 8**	**Field 9**	**Field 10**	**.....**	**Field n+8**
Temp. 1	Temp. 2	Temp. 3	Pixel #1	Pixel #2	.....	Pixel #n
16 bits	16 bits	16 bits	16 bits	16 bits	.....	16 bits

Correspondingly, there is another new STM DATA REQUEST package type, which is a down-link type for the NWPCS to request STM data from a robot. As shown in table 4-9,

the body of this type is only a single 16-bit integer which tells the target robot the maximum number of pixels (words) that can be carried in the next STM DATA package. In the current design, each STM DATA REQUEST package expects only one responding STM data segment. To get next STM data segment, the robot agent has to send a new STM DATA REQUEST package.

Table 4-9: The body structure of the STM DATA REQUEST type.

Field 6
Segment length
16 bits

Even for receiving the same STM data array, the value of the segment length in the STM DATA REQUEST type packages can be different, depending on the rest of the buffer space in the robot agent. The actual segment lengths of the responding STM DATA packages depend on how many pixels are left in the STM data buffer of a robot. If there is no STM data available, the body of the responding STM data package contains the temperature data only.

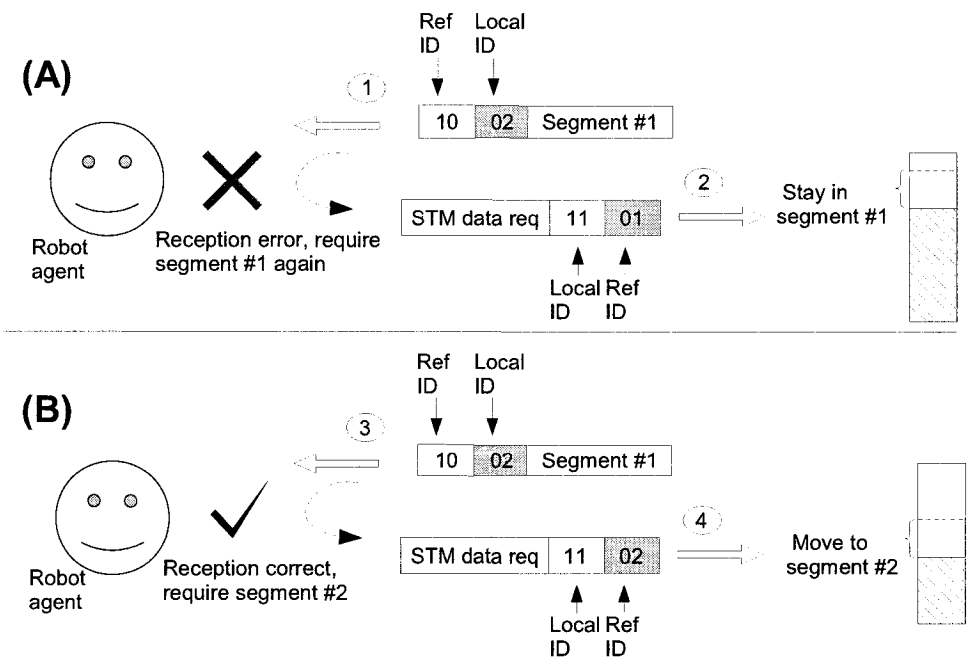

Figure 4-4: Use Reference Package ID to require the same segment or the next segment.

It is possible that a STM DATA type package received by NWPCS has error, and the robot agent will require the same STM data segment again. In that case, the **Reference Package ID** field of the next STM DATA REQUEST package will not contain the **Local Package ID** of the error STM DATA package. Otherwise, the Reference Package ID field of the next STM DATA REQUEST is the same as the local package ID of the robot, the next STM data segment will be sent out. Figure 4-4 shows an example to tell how this works.

In case **(A)** of figure 4-4, package (1) is not correctly received by the robot agent; and the robot agent sends package (2) to require STM data segment #1 again. The Reference Package ID of package (2) is different from the Local Package ID of package (1). Whereas in case **(B)** of figure 4-4, package (3) is correctly received by the robot agent; and the robot agent sends package (4) to require STM data segment #2. The Reference Package ID of package (4) is the same as the Local Package ID of package (3).

4.2.7. The modification of the status type

For the STATUS type package, its body structure is basically the same as the original proposal, as shown in table 4-2, except that to be consistent with other up-link packages, the temperature fields are now in front of the status field, as shown in table 4-10.

Table 4-10: The revised body structure of the STATUS type.

Field 6	Field 7	Field 8	Field 9
Temperature 1	Temperature 2	Temperature 3	Status
16 bits (9 bits used)	16 bits (9 bits used)	16 bits (9 bits used)	8 bits

Correspondingly, there is a new down-link **STATUS REQUEST** type, which asks the robots to present their operating status. This type has no package body.

4.2.8. The packages for displacement control

The DISPLACEMENT type is not changed too much except the reserved field is deleted. A DISPLACEMENT package can either start the movement of a robot or stop the

movement, and it needs a reply from the robot to know how the robot is going to react. The message controlling algorithm is as the following:

- If the DISPLACEMENT package has a valid body, and the robot is not doing STM scanning and moving, the robot starts to move using the parameters provided in the DISPLACEMENT package.
- If the DISPLACEMENT package does not have a body (header only), and the robot is moving, the robot stops.
- No matter the DISPLACEMENT package is requiring the robot to move or not, the robot will reply a STATUS package back to indicate what it will do next.

4.2.9. The packages for STM scanning control

The reserved fields of the STM SCANNING type and the STM PID CONTROL type are also deleted, and the rest of the types can be kept without any change, but to save the transmission overhead, in the current design, the two types are combined together, as shown in table 4-11. The new combination type is called STM CONTROL type. When a robot receives this package, it proceeds to begin the scanning.

Just as the DISPLACEMENT type, a STM CONTROL package without body is to tell a robot to stop STM scanning; and whether or not the STM CONTROL package has a body, the robot should reply a STATUS package to the NWPCS indicating what it will do next.

Table 4-11: The body structure of the STM CONTROL type.

Field 6	**Field 7**	**Field 8**	**Field 9**	**Field 10**	**Field 11**	**Field 12**	**Field 13~17**	**Field 18**
FVx inc.	FVy inc.	SVx inc.	SVy inc.	FVx up.	SVy up.	# Faxis up.	PID term 1 ~ 5	Ln (setpoint)
16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits each	16 bits

4.2.10. Summarizing the types

Currently, there are two types of up-link packages and the three types of down-link packages, as shown in table 4-10. The processes of handling the two groups of packages are completely different. To be effective in the communication, there is no specific knowledge-acknowledge scheme. Any down-link package is a knowledge call, and any up-link package is an acknowledging response.

Table 4-12: The arrangement of the types.

Down-link types		***Up-link types***	
Rank	Name of type	Rank	Name of type
0	Status request	0	Status
1	STM data request	1	STM data
2	STM control		
3	Displacement		

4.3. Conclusion and discussion

This chapter describes the structural design of the IR communication packages, and the logical relationships among different package types. The tests of the IR communication channel showed that the current design of the packages sufficiently simulated the data exchange between the robot agent simulator and the NanoWalker simulator. Despite of the success in the tests, some aspects of the design remains unclear, and need to pay more attention in the future:

- If the number of robots is more than 255, the NWID field in the package header has to be expanded to one word.
- Is it possible that each piezo-leg needs to work at different vibration frequency in the future? If yes, there should be three frequency fields in the DISPLACEMENT type.
- Currently the three words of the temperature data go along with every up-link packages and will consume some communication resources. If the design of the

cooling chamber does not need to check the internal temperatures of the robots so frequently eventually, it is better to take them off from all of the up-link types, except the STATUS type packages.

- So far there is no way to foresee how the STM is built exactly. Although the combination as a single **STM CONTROL** type can save some time in the transmission and the handling of the packages of this type, it is hard to say whether or not the combination of the STM SCANNING type and the STM PID CONTROL type is proper.

- The original STATUS field shown in figure 4-1 is not very proper. Whether or not the STM data in a NanoWalker are ready can be checked from the responding STM DATA type packages. The SDR bit in figure 4-1 can be replaced by a Moving bit, indicating whether or not a robot is moving. The revised STATUS field is as shown in figure 4-5.

Figure 4-5: The revised STATUS field in the STATUS type.

After all, designing IR communication packages is a part of designing the IR communication channel. Understanding the design of the different types of the packages is essential for understanding the design of the IR communication manager, the robot agent and the NWBOS. The next chapter gives the detailed design and implementation description of the IR communication manager.

Chapter 5. The communication manager design

The objective of building an IR communication manager is to create a package transmission mechanism that is insensitive to the content of the packages. On one side the robot agent can write down-link packages to the manager and read up-link responses from the latter without caring how the packages are transmitted; on the other side, the manager automatically transmits those down-link packages to their target robots by one at a time, and receives the responding packages in the meantime.

The communication manager is designed for controlling only one working-cell. Multiple working-cells need multiple communication managers, and the robot agent needs to coordinate the managers. For simplicity, the current version of the communication manager is not designed for multiple communication managers, although some of its parts can be easily expended for that purpose.

The concepts about Linux kernel, character device driver and user space application will be briefly explained in the later section. Before that, it is better to look at the main issues in building the communication manager.

5.1. Issues in building the communication manager

The IR communication channel consists of the IR communication hardware (IR controllers and transceivers), the FIR protocol, the communication manager (including device driver), and the portion of the robot agent that is responsible for sending and receiving packages. Among these, the communication manager is the main component to be concerned. There are a few issues to be considered in designing and implementing the communication manager:

- The reliability of the communication channel is critical to ensure the correctness of the system. If there is no strong interference to the IR signal, the FIR protocol has ensured the reliability of the communication between the IR controller of a NanoWalker and the IR controller of the NWPCS. Based on that, the reliability of handling the package transmissions is one of the major issues

in building the communication manager. Reliability requires that the design and implementation should ensure every operation of the manager is under control.

- The channel should be efficient[17] (in time) enough to meet the high throughput requirement of the system. The IR communication channel is a time-sharing channel. In other words, when the system is running, the channel is allocated to the NanoWalkers one by one within a working-cell, and each of the robots occupies the channel for a short time slot. Efficient time allocation allows the 4Mb/s FIR bandwidth to be utilized as much as possible. In this regard, another major issue in building communication manager is the time efficiency. To be time efficient, the current design of the manager avoids any error detection scheme such as the CRC to check the received packages, and the algorithms of the manager are carefully optimized (in speed).

- The design of the communication manager should care about the flexibility to the changes of the communication hardware. The bandwidth of the FIR protocol is too narrow regarding to a high throughput NanoWalker system. It is likely that a better wireless communication protocol can substitute the FIR in the future. In such case, the communication hardware controller will be replaced as well. The flexibility focuses on how to minimize the amount of codes to be changed as the hardware components are changed.

- The simplicity of the interface to robot agent is also important in the design. According to the functionality of the communication manager mentioned in chapter three, the communication manager does not have to know the detailed contents of the packages, leaving this work to the robot agent. The simplicity focuses on minimizing the number of function calls and/or variables the robot agent needs to deal with.

17 It is hard to use "fast" as a criteria of the communication manager, because faster computer, broader bandwidth can also achieve "faster" performance. From software point of view, one should aim at building "time efficient" algorithm(s) instead.

5.2. Some background knowledge about Linux drivers

The IR communication manager is designed as a character device driver running in Linux. In Linux, character device drivers are defined as kernel modules. These kernel modules are loaded and unloaded at runtime. It is necessary to give some background knowledge about the Linux kernel and its device driver before describing more tailed design of the communication manager.

5.2.1. The Linux kernel space and user space

In modern operating systems such as WindowsXP and Linux, the software environment for running kernel modules is completely separated from the one for running user application modules [20]. The former is called the **kernel space**, and the latter is called the **user space**. The separation includes not only the free memories and program codes, but also the execution modes of the CPU – the mode for running kernel programs has higher privileges. The programs in user space cannot directly access the data and functions in the kernel space.

The basic architecture of the Linux kernel can be seen in figure 5-1. The reader(s) need to keep in mind that in the figure, each functional block in fact may include many program modules. Those modules are named **kernel modules**. So far in Linux, all of the kernel modules are written in **C** language and are usually compiled with **gcc** compiler. Each kernel module consists of many functions. These functions are called **kernel functions**. As shown is figure 5-1, in Linux, the programs in the user space have to go through the functions in the System Call Interface block to access the functions and data in the kernel space; and the data to be exchanged between the two spaces have to be copied across the System Call Interface block.

Individual data can be passed across the System Call Interface as a parameter value or a return value (pass by value) of a function in the interface. Whereas, if a block of memory data has to be passed across the interface, the source and destination buffer pointers will be passed to the kernel functions. In such case, the kernel functions can use function **copy_to_user()** to export kernel space data block to the user space and use

copy_from_user() to import user space data block into the kernel space.

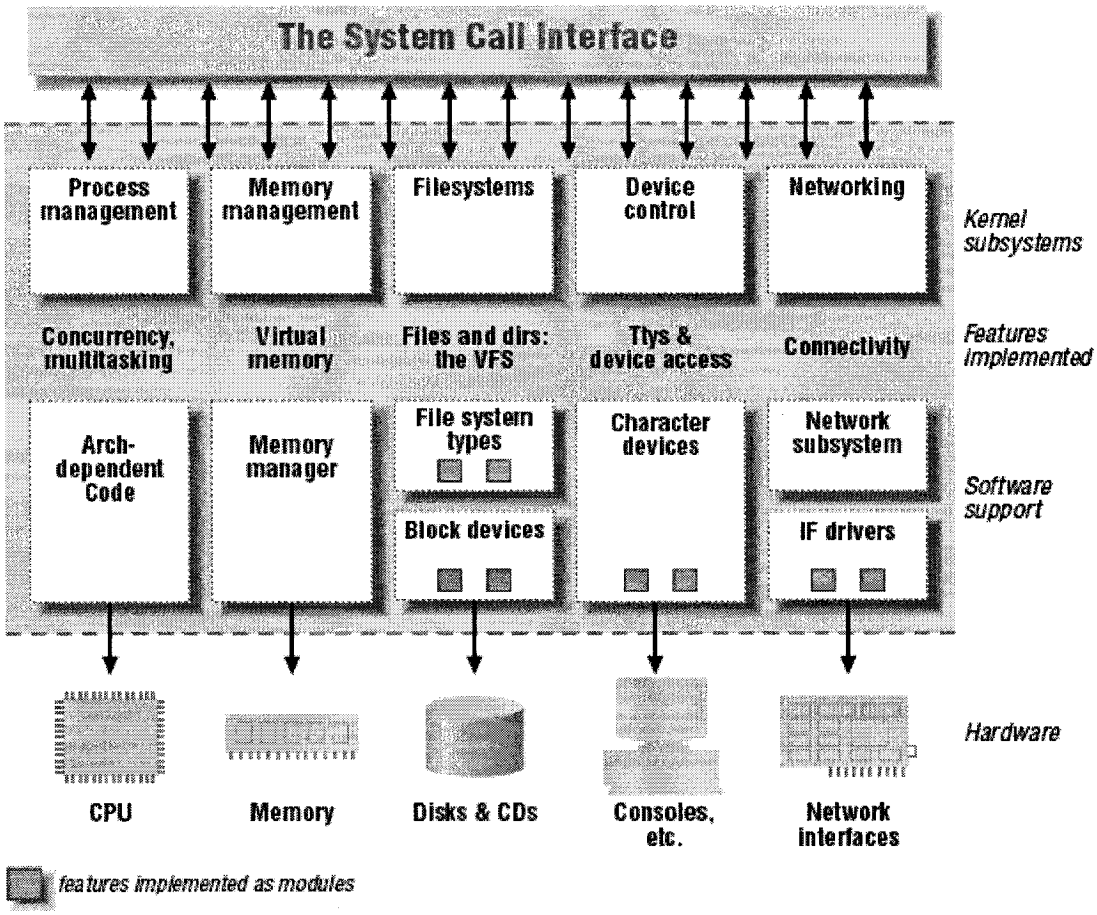

Figure 5-1: The basic architecture of the Linux kernel [20].

5.2.2. Linux kernel module and Inter-module access

One of the Linux characteristics is that it allows user defined modules to be inserted (loaded) into the kernel at runtime by command **insmod**. Every module has an initialization function[18] which will automatically execute when the operating system loads the module. This function sets up the working environment of the module such as initializing the global variables, setting up memory blocks, registering channels, acquiring ports, etc. Correspondingly, Linux provides command **rmmod** which removes a module from the kernel. When a kernel module is removed, the resources held by the modules cannot be

18 The default initialization function is named **init_module()**.

released automatically. Every module has to use a clean-up function[19] which will be called when the module is removed, to release any resource being held. Readers can refer to the Linux command manual to see how the **insmod** and **rmmod** commands are used.

Just like user space programs that can be started with command line parameters, kernel modules can be inserted into the kernel with command line parameters too. The parameters are to change the default values of some global variables in the modules. These parameters need some special declaration in the modules. In the communication manager, the actual number of robots in the system, the capacities of the incoming buffer and outgoing buffers, the length of the time slots, and the maximum package length (will be mentioned later) are the parameters that can be changed during insertion.

5.2.3. The kernel symbol table

The Linux kernel is a single layer structure. It allows its internal functions to fully access any resources and can even replace some important functions originally coming with the Linux distribution. Although the kernel modules have the privilege to fully control the behaviour the operating system, they cannot see each other. In other words, they cannot see the global variables and functions of other modules directly. The purpose of preventing kernel modules to see each other directly is to reduce the name conflicts among modules, especially for an operating system such as Linux which is developed by a loosely organized community.

Instead, the Linux kernel uses a special **kernel symbol table** for the modules to register their function names and public variable names which are desired to be seen by all modules in the kernel. The kernel symbol table can be seen by all kernel modules. In any kernel module, the function names and variable names to be seen by other modules have to be registered in the kernel symbol table with a special declaration method[20].

The drawback of using the kernel symbol table is that the declarations and registrations are relatively complicated. As will be mentioned later, in the implementation, the

19 The default clean-up function is named **cleanup_module()**.

20 Unfortunately there is no guarantee that the declaration method is the same across different versions of Linux kernel. The current version of Linux kernel used in the implementation is 2.4.22.

communication manager is separated into three parts. Besides the consideration of separating the functionalities, how to minimize the number of symbols that need to be registered in the kernel symbol table is another consideration when thinking about the organization of each part.

5.2.4. The device drivers

The number of functions, the type of functions and the signatures of those functions in the System Call Interface are strictly determined by the Linux kernel structure, the feature of the intrinsic kernel modules and the architecture of the user space libraries. Any change in the System Call Interface may seriously affect the structure of the whole operating system. Therefore the structure of the System Call Interface is usually not subject to be changed. It is not proper to add new functions into the System Call Interface for the programs in the user space to utilize the newly inserted kernel modules. Instead, the easiest way for the programs in the user space to use those kernel modules is to define the user defined kernel modules as **device drivers**.

In Linux, device drivers are managed by the device control unit and are considered as part of the kernel, as shown in figure 5-1. As kernel modules, the user customized device drivers can be inserted (loaded) into the Linux kernel at runtime.

Linux device drivers have three different types namely the **character type**, the **block type** and the **network interface**. The communication manager is designed as a character type device. The advantage of being a Linux character device driver is that it can fully manage the (kernel) memory allocation and its own execution time. Another advantage of being device driver is that kernel modules respond faster than the user space applications in general.

The character type devices are treated as normal data files. They can be "opened", "read", "written" and "closed" just as a text file. Linux provides five functions to the user space programs to manipulate data files and character devices: **open()**, **close()**, **write()**, **read()** and **ioctl()**. As an example, figure 5-2 shows the calling relationships between the user space file manipulation functions and their corresponding functions in the

communication manager. In figure 5-2, the System Call Interface directs these user space function calls to the corresponding device-open, device-close, device-write, device-read and device-I/O-control functions[21] inside the device drivers. The latter are defined by the developer of the devices and are registered in a file operation structure which is provided by Linux. The device-I/O-control function is used for changing the working status of the devices or passing some new values to the global variables in the modules.

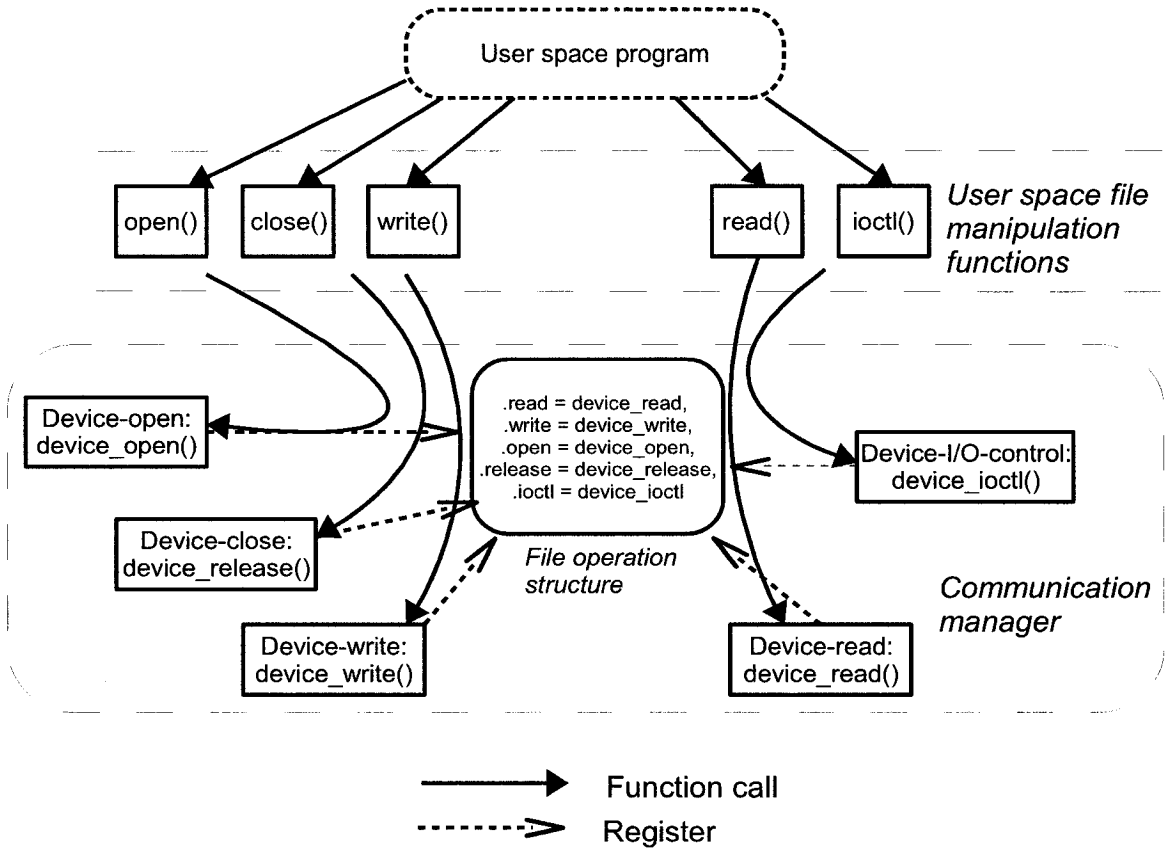

Figure 5-2: Function call relationships between the user space and the communication manager.

It should be noted that a device driver may include multiple kernel modules, but only one module can be registered as a device to the Linux. The functions and global variables are shared by the modules with the kernel symbol table. For more details about how to register a kernel module as a device as well as how to use the above-mentioned device manipulation functions, one can refer to the device programming handbook and the user manual of Linux.

21 In the communication manager, these functions are named **device_open()**, **device_release()**, **device_write()**, **device_read()** and **device_ioctl()** respectively.

A question should be noticed here: can the user space applications use the file management function **fwrite()** and **fread()** in standard C library or the **iostream** class in standard C++ library to write packages to and read packages from the communication manager? Well, in fact, the first student who wrote the package sender part of the robot agent did use the **iostream** to write packages to the communication manager. Whereas the later tests showed that packages longer than about 350 bytes could not be completely sent by the communication manager. After a lot of testing and debugging, it was realized that the file management functions in standard C and C++ are not compatible with the file management functions supplied by Linux completely. Only function write() and read() in Linux can safely send unlimitedly long data streams to the character devices (as long as the devices can handle).

5.2.5. Wait_queue and timer task

Just like in the user space, processes of the kernel functions can also be put to sleep while waiting for some resources to be available or some other events to occur. Linux provides a structure **wait-queue** to hold the sleeping processes. After a process is put to a wait-queue, it is suspended (put on sleep) from being scheduled for execution until an expected event occurs. Each wait-queue can hold a set of processes that wait for the same event. When the event occurs, the event handling routine wakes up the wait-queue, and all processes in the wait_queue will be activated.

A timer task is a task associated with a kernel timer mechanism. It is a separate task other than the one that initializes it. It is used for scheduling an execution of a registered function at a future time. In this thesis, the function registered to a kernel timer is called the service function of the timer. It is executed only once when the preset time of the timer is up, but can be rescheduled to the timer task again. Before the expected time is up, the timer task is put to sleep by the Linux kernel, and the programmer of the device driver does not have to care about how to suspend it and wake it up later.

The Linux kernel uses a time click mechanism, **jiffies**, instead of the real time clock to express and calculate time. Each click is called one **jiffy**. To set up a kernel timer, the expected **jiffies** value is registered to the timer. For example, suppose the current system

jiffies is 1000, and the timer is to invoke the service function 15 **jiffies** later, then the expected time period to be registered to the timer here is 1015 (**jiffies**). The length of one **jiffy** is calculated as 1/**HZ**, where **HZ** is a Linux system constant and is platform dependent. For PC computers, the value of **HZ** is 100, and thus one **jiffy** is 10ms long.

As will be mentioned later, the communication manager uses a wait_queue and a kernel timer task forms the mechanism of allocating time-slots to the robots.

5.3. The general structure of the communication manager

Starting from this section, the detailed design of the communication manager will be described. The original intention is to make the communication manager as a user space program, just like other managers in the NanoWalker system, which utilizes a device driver which directly controls the hardware operation of the PC87108A. However such architecture cannot fully control the memory allocation and the time scheduling of the modules. The current design is to let the communication manager itself to be a device driver with "extended" features that are far more than simply controlling the hardware operations. From the six-layer architecture point of view, one cannot say that because the communication manager is designed as a device driver in Linux, it belongs to the driver layer. In fact as seen in the next section, only the direct hardware controlling part of the manager belong to the driver layer, and the rest of the manager belongs to the manager layer.

5.3.1. Logical design of the communication manager

The communication manager can be divided into 11 functional components as listed here:

1. Device-write function, as mentioned in section 5.2.4, an interfacing function corresponding to the file write() functions in the user space;
2. Device-read function, as mentioned in section 5.2.4, an interfacing function corresponding to the file read() function in the user space;
3. Package-import, a routine for importing down-link packages from the robot

agent to the outgoing buffer(s);

4. Package-export, a routine for exporting up-link packages from the incoming buffer to the robot agent;

5. A number of memory buffers, including one incoming buffer and set of outgoing buffers for holding up-link and down-link packages respectively;

6. Timed-package-send, a routine for sending out the down-link packages in the outgoing buffer to the robots, including a time scheduling routine for allocating time slots to robots so that the sending routine can send the packages of the current robot;

7. Receiving-interrupt-handler, a routine for handling the receiving DMA operation of the IR controller which writes the incoming packages to the incoming buffer directly;

8. Hardware controlling part of Timed-package-send, as implied by its name, responsible for controlling the infrared data frame transmission operations of PC87108A;

9. Hardware controlling part of Receiving-interrupt-handler, similar to the previous one, responsible for controlling the PC87108A controller to receive infrared data frames;

10. Device-I/O-control function, an interfacing function corresponding the file ioctl() function in the user space,

11. Finally, the basic module control portion for initializing (when loading communication manager), cleaning up (when unloading the communication manager) and everything else that are not shown in figure 5-3.

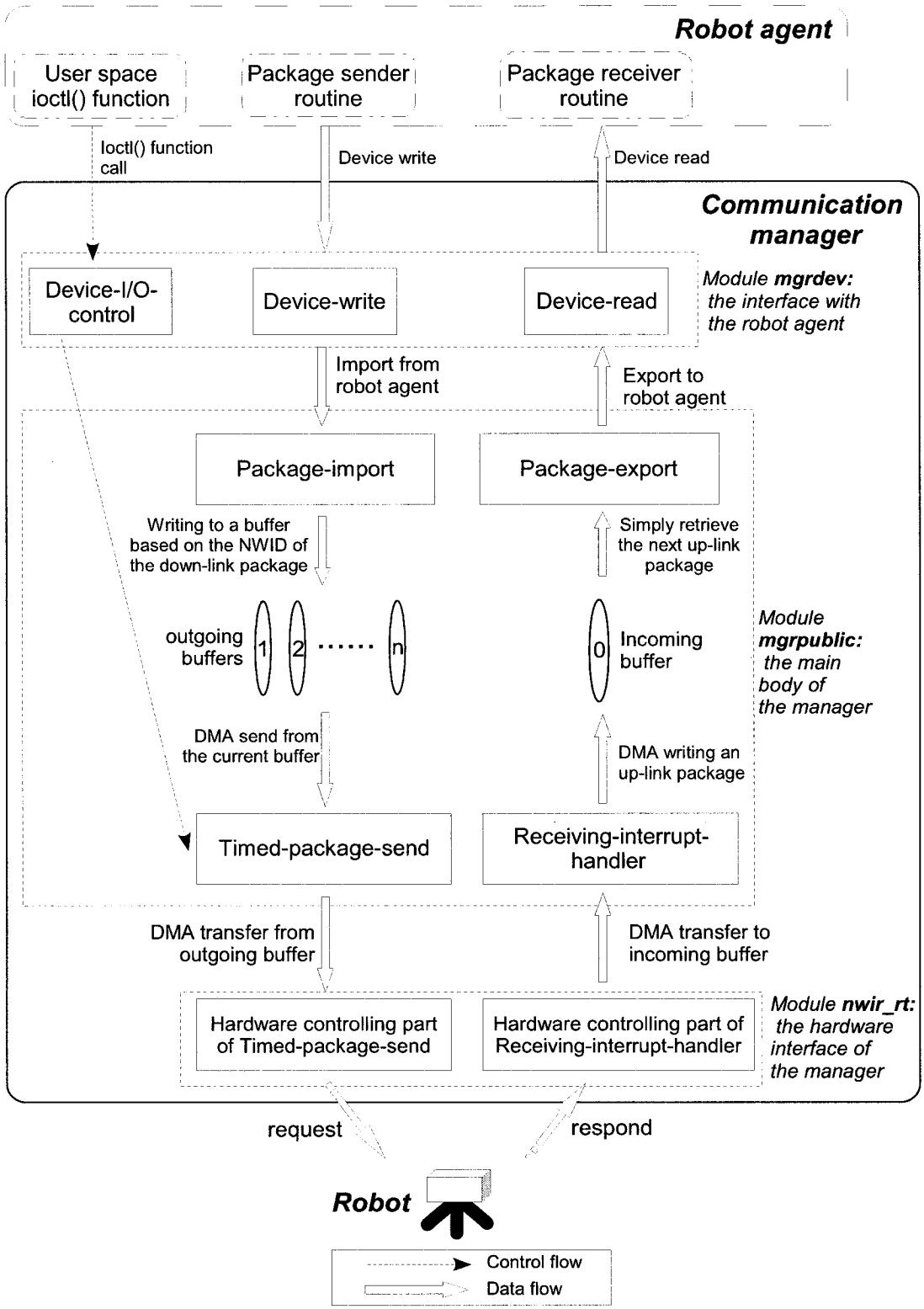

Figure 5-3: The structure and logical relationships inside the communication manager.

The logical relationships of the components are depicted as in figure 5-3. The package sender and receiver routines at the top of figure 5-3 do not belong to the communication manager.

The device-write and device-read routines are very simple functions which mainly pass the user space buffer pointers and other parameters to the package-import and package-export routines respectively. Package-import and package-export routines may change in the future. If the structure of the packages changes significantly, separating device-write from package-import and device-read from package-export will allow a stable interface structure to link with the user space file manipulation functions.

The device-ioctl function can be used for changing the runtime parameters in the communication manager. In the current implementation, it is mainly for manually controlling the timed-package-send routine, as will be described later.

The two hardware controlling parts at the bottom of figure 5-3 actually belong to the two routines above them respectively. The reason for separating them from the above is that they control the hardware (PC87108A and the DMA controller of the ISA bus) operations directly.

5.3.2. The organization in the implementation

Although the communication manager is defined as one device driver, it is in fact divided into three kernel modules. The first module, namely **nwir_rt**, contains the hardware interface portion of the communication manager, including the hardware controlling part of the timed-package-send routine and the hardware controlling part of the receiving-interrupt-handler routine. The second module, namely **mgrpublic**, is the main module of the manager, including the package-import, package-export, timed-package-send and receiving-interrupt-handler routines as well as the incoming and outgoing buffers. The third module, namely **mgrdev**, is the interface module with the user space, containing the device-write, device-read and device-ioctl functions. Functions and variables in one module can be used by another module after being registered to the kernel symbol table.

The reason for separating module **nwir_rt** from module **mgrpublic** is to allow the hardware interface portion to be updated more easily in case the PC87108A is replaced by another communication hardware controller in the future. Similarly, the reason for separating another module **mgrdevice** from module **mgrpublic** is to allow the user space interface to be changed easily when necessary.

5.3.3. Implementing kernel module nwir_rt

The **nwir_rt** module was originally developed by an undergraduate student [21]. The original version of **nwir_rt** was in fact like a normal character device driver and was intended to be used by the programs in the user space only. It mainly included the following functions:

- init_module() -- initializing the module itself and the hardware registers when the module was load by the operating system;
- nwir_rt_read() -- when a user space program read the device, this function was responsible for outputting the data stored in the receiving (incoming) buffer of the device to the user program;
- nwir_rt_write() -- when a user space program writes some data to the device, this function was responsible for inputting the user data into the sending (outgoing) buffer of the device;
- cleanup_module() -- cleaning up the (memory occupied by the) module itself when the module was unloaded by the operating system.

The problem of the original version was that it had no mechanism to allow other kernel modules to access directly; the operations for the hardware registers are not optimized; and most importantly, the receiving of the communication data did not work properly – the data could not be received completely. When this module was taken over, the first thing was to spend a lot of time to debug the data receiving problem. Eventually, it was realized that the problem was because by mistakenly using the **Receiver High-Data-Level Event** of the **Event Identification Register** (when **RXHDL__IE** bit in **Interrupt Enable Register** is set) as the interrupt signal to trigger the receiving-interrupt-handler routine. This wrong

interrupt signal caused the interrupt handler to shut off the DMA channel unexpectedly right at the beginning of receiving. The correct interrupt signal should be the **DMA Status FIFO Event** of the **Event Identification Register** (when **SFIF_IE** bit in **Interrupt Enable Register** is set), which signifies the end of DMA transmission of the data from the rx-FIFO of the PC87108A IR controller to the memory buffer of the device driver.

The above-mentioned signals and registers are the properties belonging to PC87108A. A side-effect of using **Status FIFO Interrupt Event** is that the PC87108A cannot receive multiple packages continuously. In other words, the NWPCS has to use the one-sent-one-received scheme to communicate with the NanoWalkers, and if two packages are received continuously, the first one is always missed. This is because the design of the PC87108A enforces that the **Status FIFO Event** signal can only be generated by every other DMA operation.

Other changes to the original **nwir_rt** module include the optimization of the codes, disabling the read() and write() functions, adding interrupt handlers for sending, etc.

As will be mentioned later, the timed-package-send routine also has an interrupt handler routine. In the revised version of module **nwir_rt**, this handler routine combines with the receiving-interrupt-handler routine in one function.

5.4. The design of the package buffers

As shown in figure 5-3, in current design, each communication manager has only one incoming buffer for receiving all packages from the robots in the working-cell, and many outgoing buffers. The number of outgoing buffers is the same as the number of robots on the working platform. In other words, each robot on the platform is corresponding to one outgoing buffer.

5.4.1. The buffers structure

Each package buffer is a circular queue. A circular queue has the benefit that when the package at the top of the queue is removed, the rest of the packages in the queue do not have to be moved up, saving the time of moving memory data. The drawback of a circular queue is that the data structure is more complicated than that of a simple queue.

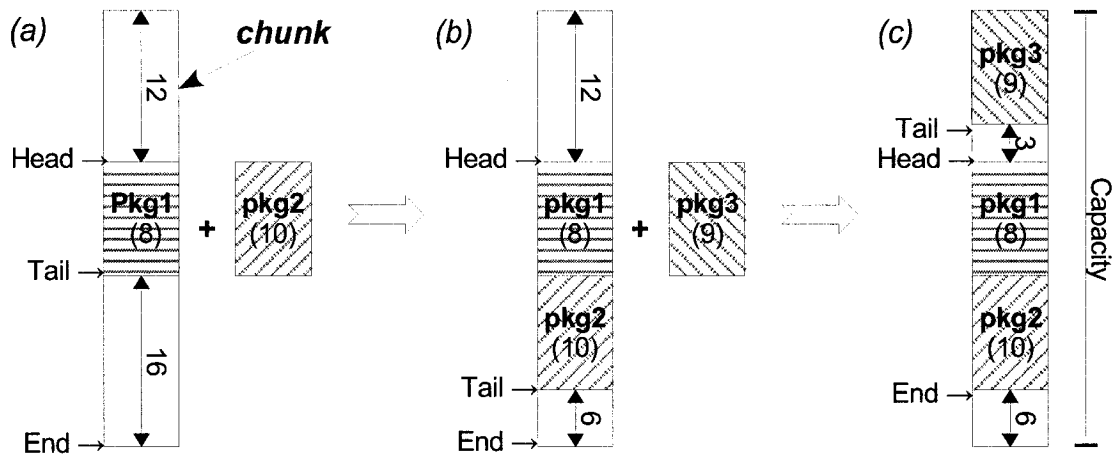

*Figure 5-4: Using a dynamic **End** pointer to ensure all packages are continuous.*

The circular queue used in IR communication manager includes a memory area, a **Head** pointer, a **Tail** pointer, a **Capacity** value, and a dynamic **End** pointer which indicates the "dynamic" end of the queue. In other words, the queue does not always occupy the whole memory area. To distinguish, the memory area is called **chunk**. Because the DMA operations require that the data to be transferred have to be continuous in memory address, the "dynamic" end of the memory array ensures that the packages will not be "cut" in the middle. Figure 5-4 shows how the **End** pointer works. In case (a) of figure 5-4, the **End** pointer is at the bottom of the chunk and the space from **Tail** to **End** is enough for adding the package 2. In case (b) of figure 5-4, after package 2 is added, the space from **Tail** to **End** is not enough for adding package 3. In such case, as shown in case (c), package 3 is added at the top of the chunk (the space from the top of the chunk to Head is enough for holding package 3), the **End** pointer is at the end of package 2, and the **Tail** pointer is at the end of package 3. One can see that in case (c), the queue dose not occupy the whole chunk.

5.4.2. Distinguishing an empty circular queue and a full circular queue

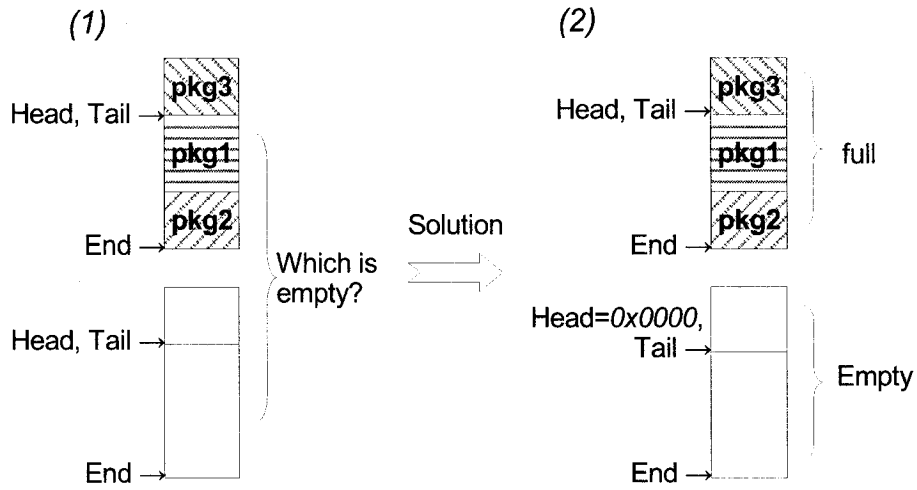

Figure 5-5: Distinguishing empty circular queue and full circular queue.

In practice, the **Tail** pointer always points to the next element right after the end of the last package. If the queue is not full, this position is the beginning of the next package. If the queue is full, the **Tail** pointer is the same as the **Head** pointer. The problem is, when the queue is empty, the **Head** pointer is also the same as the **Tail** pointer. How to distinguish if the queue is empty or full? The solution is that when the queue is empty, let the **Head** pointer pointing to NULL (0x0000), and if a package is added to the empty queue, **Head** is restored to the value of **Tail** before setting **Tail** to a new value. Figure 5-5 shows the idea.

5.4.3. Maximum package length and full state of the packages

The maximum package length means the maximum length of a package that the communication manager can encounter possibly. As one can see in chapter four, the STM DATA packages are usually the longest ones, hence the maximum package length in fact means maximum possible length of this type of packages. It should be decided by the programmer by default, based on large number of tests to the communication manager. On one hand the maximum package length needs to be as long as possible so that the over head for processing the transmission of the packages is to be as low as possible. On the other hand, as can be seen in the later paragraphs, the longer the maximum package length is, the lower the efficiency of using the incoming buffer will be. To be flexible, the maximum package length is declared as a command line parameter which can be set to a new value when loading the communication manager.

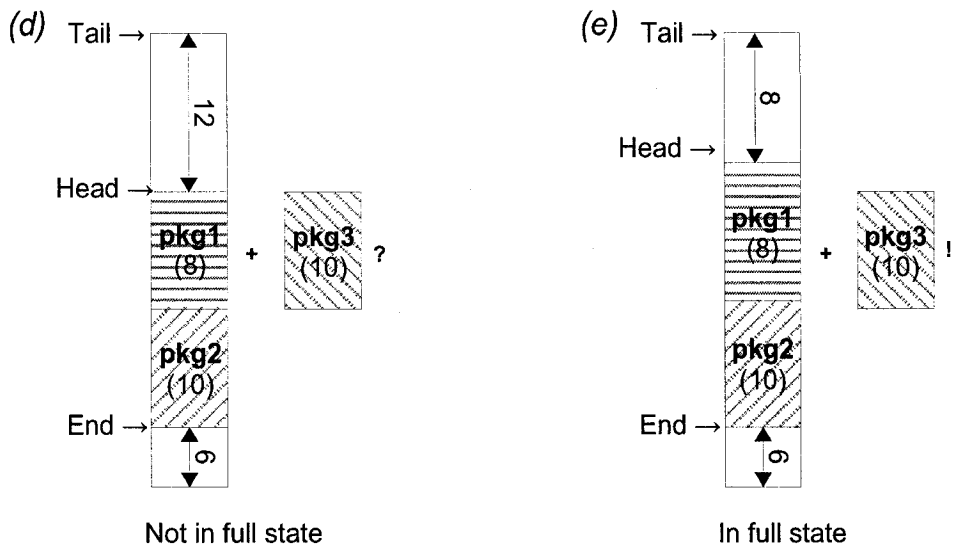

Figure 5-6: Distinguishing the full state of a circular queue.

One use of the maximum package length is to determine whether or not the **Tail** pointer of the circular queue needs to be put at the top of the chunk. For example in case (b) of figure 5-4, assume that the maximum package length is 10 for example, because the communication manager does not know exactly the contents of the packages and it has no way to estimate what kind of up-link package it will receive, when preparing the DMA buffer for the next receiving, it always assumes that the next incoming package takes the maximum length. Now the space between **Tail** and **End** is 6, which is less than the maximum package length. To receive next up-link package, **Tail** has to be set to the top of the chunk and **End** is set to the end of package 2 as shown in case (d) of figure 5-6.

The space between **Tail** and **Head** is 12 in case (d), which is greater than the maximum package length. Therefore case (d) is safe for receiving next up-link package. If the space between Tail and Head is less than the maximum package length, as shown in case (e), because the next up-link package may over write the content of package 1, the communication manager should consider that queue is full, and the incoming buffer is now said to be in **full state**. The full state means that a circular queue in the communication manager does not have enough space to hold that next package. This case can happen in the outgoing buffers too.

From the analysis above one can see, as the maximum package length increases, the

percentage of the unusable space in case (e) of figure 5-6 increases too. Therefore the drawback of very long maximum package length is that the usage of the buffer space is not efficient. If the system has a large number of robots, each communication manager will have large number of buffers, and such an inefficiency will become significant.

5.4.4. The buffer sizes

The communication manager needs to prepare the buffers for all possible robots in the system. These buffers are allocated and initialized when the communication manager is loaded into the kernel. The default size of the incoming buffer chunk is set to two pages[22], and the default size of each outgoing buffer chunk is one page. For one hundred NanoWalkers, for example, the total amount of memory needed by the buffers will be 408k (417792) bytes.

According to the current design of the piezo-legs, it can be expected that the longest down-link package (which should be a DISPLACEMENT type package) is less than 500 bytes. The default maximum package length in the current implementation is 1024. Therefore by default the minimum number of packages that the outgoing and incoming buffers can hold is at least eight. As being tested so far, such default buffer sizes are proper with regard to the total size of the memory of the PC computer and package transmission speed of the communication manager.

The default sizes can be increased, of course. If the processing speed of the NWPCS is slow, then the eight-package tolerance may not be enough for buffing the package traffic. In that case, the sizes of the buffers need to be enhanced, and the total amount of memory allocated to the buffers will increase proportionately.

5.4.5. Buffer allocations in the implementation

In current implementation, the buffers are defined as the objects of structure **chunk**. The structure includes the **Head**, **Tail** and **End** pointers, the pointer to the circular queue, the capacity of the chunk, as well as a semaphore that is used for synchronizing the operations to the pointers. The buffers are allocated as one series of **chunk** objects.

---

22 In Linux, one page of memory is 4k (4096) bytes.

The first buffer (buffer zero) is used as the incoming buffer; and the rest of the buffers are assigned to the NanoWalkers with the corresponding NWID respectively, i.e. buffer #1 is assigned to NanoWalker #1, and buffer #2 is assigned to NanoWalker #2, etc. Such an arrangement is simpler, faster and more intuitive than other type of arrangements (for example using a linked list to hold randomly assigned buffers).

It is possible that the default buffer sizes are too large to fit the memory of the PC computer. The current allocation algorithm uses a dynamic method to solve such a problem: if the allocation routine (function **setupChunks()** in module **mgrpublic**) fails, it will automatically reduce the size of the buffers until all buffers can be allocated successfully. If the sizes are reduced to less than allowable limits, the loading of the communication manager fails, and the PC computer need to be upgraded with more memory. The default buffer sizes and their lower limits are parameters of module **mgrpublic** that can be changed during loading.

5.5. Synchronizing import, export, sending and receiving

No matter how big the capacities the package buffers have, they are still possible to get to a full state. If the incoming package is in full state, the next up-link package received by the PC87108A is lost, or one of the packages in the buffer is overwritten. Fortunately the message exchanging scheme between the NWPCS and the NanoWalkers is a one-request-one-response style. If the communication manager does not send a down-link package, it should not receive any up-link package. Therefore the communication manager uses an integer flag **inOK** to synchronize the IR sending and receiving operations: if the incoming buffer is in full state, the value of **inOK** is zero, otherwise it is one. Before the Timed-package-send routine sends a down-link package, it checks the value of **inOK** – if it is zero, the package is not sent.

The full state of the incoming buffer is checked by Receiving-interrupt-handler routine. After finishing adjustment of the **Head**, **Tail** and **End** pointers, the routine will check if the incoming buffer is in full state – if yes, **inOK** is set to zero. On the other hand, the package-export routine is responsible for setting **inOK** to one if the incoming buffer is not in full state after exporting an up-link package.

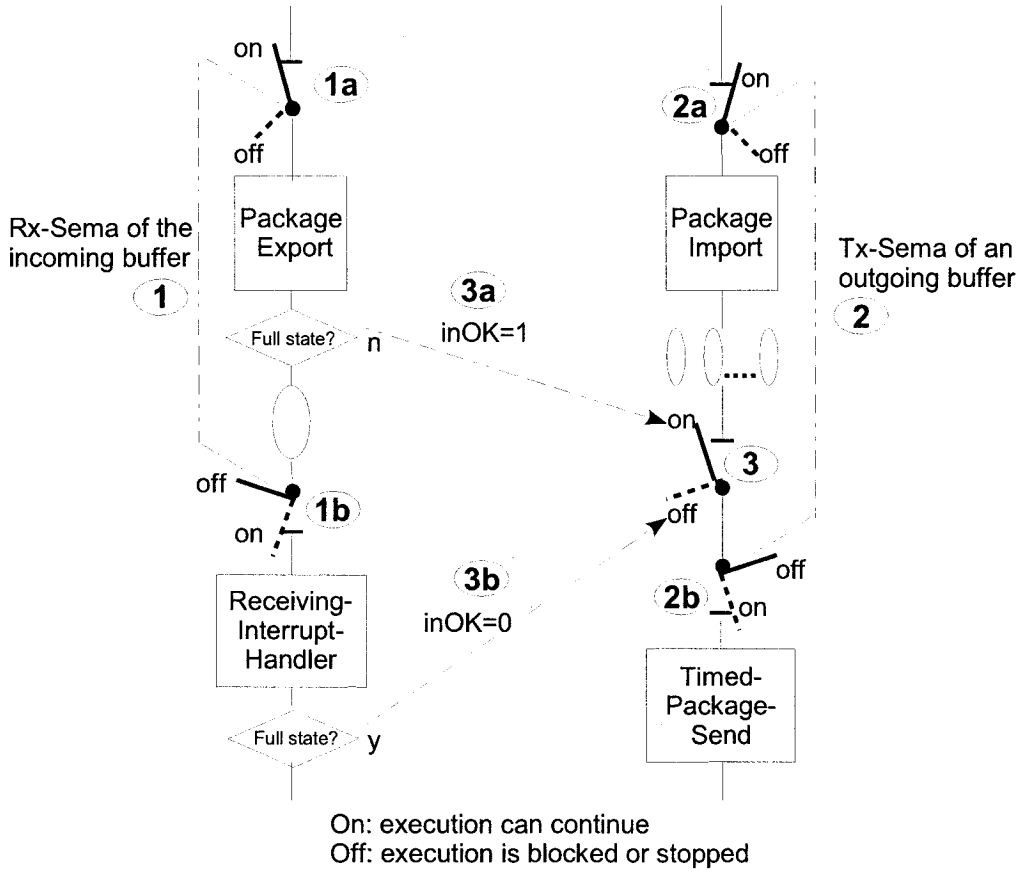

Figure 5-7: The three synchronization logics of the four routines.

The Receiving-interrupt-handler routine is invoked by the DMA controller of PC87108A. It belongs to a different task from that of the package-export routine which is invoked by the robot agent. Since both of them can change the value of **inOK**, the communication manager needs a semaphore mechanism to prevent them from writing to **inOK** simultaneously. In addition to **inOK**, the **Head**, **Tail** and **End** pointers of the incoming buffer are protected by the semaphore mechanism too. Similarly, the communication manager also uses a semaphore mechanism to protect the **Head**, **Tail** and **End** pointer of the current outgoing buffer[23] from being changed by the package-import and timed-package-send routines simultaneously.

Figure 5-7 shows the three synchronization logics of the four routines. The three synchronization logics are: rx-semaphore of the incoming buffer (1), the tx-semaphore of the outgoing buffer (2), and the **inOK** flag (3).

23 **Current outgoing buffer** means the outgoing buffer that the current time slot is allocated to. The same meaning of this term applies to the rest of the thesis.

With respect to the rx-semaphore of the incoming buffer, figure 5-7 means that if the package-export routine is holding the semaphore to change the **Head**, **Tail** and **End** pointers of the incoming buffer, then the receiving-interrupt-handler routine has to wait at (1b), the beginning of the handler, until the semaphore is released (so that it can change the pointers). Contrary, if the receiving-interrupt-handler is holding the semaphore, the package-export routine has to wait at (1a), right before changing the pointers, until the semaphore is released. In the actual implementation, since the receiving-interrupt-handler is already running in the **interrupt** mode, it does not need to hold the rx-semaphore. The interrupt mode does not allow any semaphores appear in the handler routines, and guarantees that other routines will not run at the same time with the handlers. On the other hand, the package-export routine obtains the rx-semaphore with the **down()** function[24] so that it cannot be interrupted before the semaphore is released.

Similarly, the timed-package-send routine, as will be mentioned later, belongs to a different task other than that of the package-import routine which is invoked by the robot agent. For the communication manager, every outgoing buffer associated with a tx-semaphore (2) which synchronizes the writing procedure to the **Head**, **Tail** and **End** pointers by the two routines. With regard to the tx-semaphores of any outgoing buffer, figure 5-7 means that if the package-import routine is holding the semaphore of the current outgoing buffer to change its **Head**, **Tail** and **End** pointers, the timed-package-send routine has to wait at (2b), right before changing the pointers, until the semaphore is released. If the timed-package-send is holding the semaphore of the current outgoing buffer, and the package-import routine wants to add a new package into the current outgoing buffer, the package-import routine has to wait at (2a), right before changing the pointers, until the semaphore is released. This time, since the pointer handling portions of the two routines are not running in interrupt mode, they obtain the tx-semaphore with the **down_interruptible()** function.

With regard to the **inOK** flag, if the incoming buffer is not in full state at the end of the

24 The down() and down_interruptible() functions are two kernel functions of Linux. The routine cannot be interrupted if it is holding a semaphore obtained with the down() function. On the other hand, if the semaphore is obtained with the down_interruptible() function, the routine can be interrupted. See chapter six of *LINUX Device Driver* [20] for more details.

package-export routine, then the value of **inOK** is one as shown at (3a) of figure 5-7, and the timed-package-send routine can work freely; if the incoming buffer is in full state at the end of the receiving-interrupt-handler, then the value of **inOK** is zero as shown at (3b), and the timed-package-send routine will quit at (3), which is before sending out the package(s).

In the implementation, rx-semaphore and tx-semaphores are in fact just the same semaphore which is defined in structure chunk as introduced in section 5.4.5. The **inOK** flag is defined as a global variable in the **mgrpublic** module. All of the **Head**, **Tail** and **End** pointers as well as the **inOK** flag are declared as volatile variables to make sure that the changes to these variables in one task can be seen by other tasks immediately.

After all, the synchronization problem among the four routines is one of the important factor for ensuring the reliability of the communication manager and needs a lot of background knowledge about how the time-sharing architecture works in Linux.

5.6. Sending packages with timed control

The Timed_Package_Send routine has two issues: one is the allocation of time slots, and the other is DMA transmission of packages. In general, it consists of a timer task and its service function, a sending routine and two wait_queues, a robot selector (variable). The hardware controlling part of the sending routine includes the DMA transmission interrupt handler routine. The structure of the routine and the control logic amount the components are as shown in figure 5-8.

5.6.1. Allocating time slots

As shown in the upper part of figure 5-8 the kernel timer and its service function, **wait_queue 1**, the robot selector, the final wake up call of the package-import routine make up of the time slot allocation mechanism. These components are mainly located in the **mgrpublic** module, including the DMA transmission interrupt handler, is located in the **nwir_rt** module, except that the hardware part of the sending routine. Figure 5-8 also shows the control flow chart diagrams of the sending routine and the timer service function. The reader of the thesis should keep in mind that the timer service function, the sending routine, and the package-import routine are running in different processes independently.

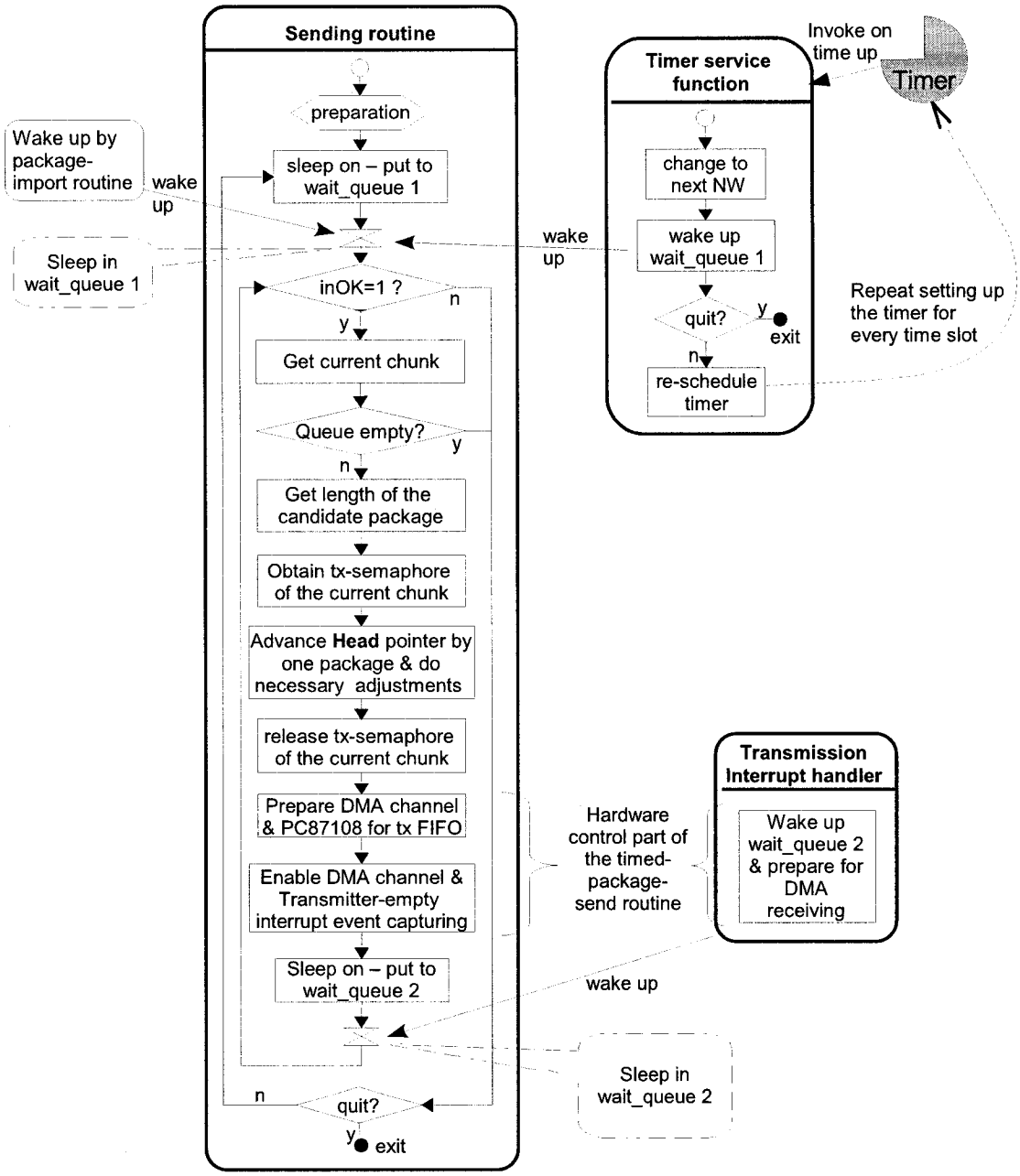

Figure 5-8: The control flow diagram of the timed-package-send routine.

The kernel timer is initially started and stopped by the device-ioctl function in the **mgrdev** module. The device-ioctl function registers the expected time **jiffies** to the timer. Figure 5-8 shows that when time is up, the kernel timer invokes the timer service function. The service function will then set the robot selector to the next robot and wake up **wait_queue 1**. The final step of the timer service function is to re-register a new expected **jiffies** value to the timer for the next scheduling. This step is important for keeping the timer work continuously.

The sending routine is started by the device-ioctl function in the **mgrdev** module too. As shown in figure 5-3, the device-ioctl function is invoked by the ioctl() function in the user space. In fact it is supposed to be started as a separate task other than the one that the robot agent belongs to. When the routine is started, it puts itself to **wait_queue 1** immediately. After it is woken up, if the incoming buffer is not in full state (the value of **inOK** is 1), and the current outgoing buffer is not empty, it starts sending package(s) from the current outgoing buffer to the target NanoWalker. The sending routine will keep sending out the down-link packages in current outgoing buffer until it is empty, or the robot selector is changed to the next robot. After all of the packages in the current outgoing buffer are sent out, the sending routine will loop back to its beginning and change to sleep status again.

A problem is encountered here: how the package sending operation switches from the current time slot to the next time slot, if the sending routine is not in **wait_queue 1**? This situation happens if the current outgoing buffer have multiple packages to be sent out, when the current time slot is finished, there are still package(s) left in the buffer. At such time, the kernel timer invokes its service function, and the service function sets the robot selector to the next robot. Because the sending routine is not sleeping in **wait_queue 1**, the wake-up call raised by the service function has no effect. The sending routine will finish sending the current package (if still under processing), and loop back to "get current chunk" step in figure 5-8. Now the robot selector is pointing to another NanoWalker, and the "current chunk" that the sending routine can get (in that step) belongs to the new robot too. Therefore that operation of sending package automatically switches from one time slot to the next without going to **wait_queue 1**.

5.6.2. Waken up by the package-import routine

In addition to being waken up by the timer service function, the **wait_queue 1** may also be waken up by the package-import routine, as shown at the left side of the figure 5-8. It is possible that a new down-link package is added to the current outgoing buffer after the sending routine sends out all existing packages. If the current time slot is not finished yet, the new package should be sent out as well. In such case, since the sending routine is sleeping again, and the kernel timer service function will not wake up the wait_queue any more until the next time slot comes up, the package-import routine has to wake up the

wait_queue by itself at the end of importing the new package.

5.6.3. Sending out down-link packages

The lower part of figure 5-8 shows the mechanism of the DMA transmission of packages. The send routine uses DMA to transmit the content of the candidate package to the tx-FIFO buffer of the PC87108A. Data transmitted to the tx-FIFO buffer are sent out to the IR transceiver immediately. The whole DMA and IR transmission tasks cannot be controlled or monitored by the communication manager. The communication manager can only provide the address of the outgoing package to the DMA controller (in the PC computer). It is the PC87108A which monitors the status of the tx-FIFO and generates an end-of-transmission interrupt when the IR transmission ends. The event to be captured by PC87108A for the end-of-transmission interrupt is **transmitter-empty**. Since the IR transmission speed is relatively low compared with the CPU processing speed, the sending routine has to sleep for a while to wait for the completion of DMA transmission. For this purpose, the sending routine puts itself to the second wait_queue, **wait_queue 2**, after it enables DMA transmission. The transmission interrupt handler routine wakes up the sending routine when the **transmitter-empty** event rises.

The detailed algorithm of sending out a down-link package is as the following steps:

1. (Starting from being waken up from **wait_queue 1**) if the incoming buffer is in full state, do not send any down-link package, go to sleep in **wait_queue 1** again.
2. Get the current outgoing buffer indicated by the robot selector variable.
3. If the current outgoing buffer is empty, go to sleep in **wait_queue 1** again; otherwise find the candidate down-link package pointed by the **Head** pointer of the current chunk.
4. Find out the length of the candidate package indicated by its **Package Length field**.
5. Obtain the tx-semaphore of the current outgoing buffer with the

down_interruptible() function to protect the **Head**, **Tail** and **End** pointers. At that moment if the tx-semaphore is being held by the package-import routine, the sending routine will be put to sleep until the semaphore is released.

6. If the queue becomes empty now, **Head** is the same as **Tail**. In such case, let **Head** points to 0x0000. However, if the queue is not empty yet, now the **Head** is pointing to the beginning of the next up-link package.

7. Adjust the **Head** pointer to the element right after the end of the candidate package. If the **Head** is less than **Tail**, one can say that the queue is in **forward case**; otherwise one can say that it is in **inverse case.**

8. In inverse case, if the queue is not empty and the **Head** pointer is the same as the **End** pointer after the adjustment, it means that the circular queue should be set to forward case. To set the queue to forward case, the **End** pointer is put back to the end of the chunk, and the **Head** pointer is further set to the beginning of the chunk.

9. Release the tx-semaphore of the current outgoing buffer.

10. Set up the PC87108A for IR transmission.

11. Prepare the DMA channel on both the PC computer side and the PC87108A side for transferring data to the tx-FIFO.

12. Enable the DMA channel and end-of-transmission interrupt capturing.

13. Go to sleep in **wait_queue 2** to wait for the completion of transferring the package.

14. After waken up by the transmission interrupt handler, loop back to step 1 above.

15. At the meantime, the transmission interrupt handler routine prepares and enables the DMA channel and the end-of-reception interrupt capturing for IR receiving.

Step 10, 11, 12 and 15 belong to the hardware controlling part of the timed-package-send routine, as shown in figure 5-3, because they directly deal with the registers in

PC87108A and the DMA controller of the ISA bus.

5.6.4. Controlling the timed-package-send routine

As introduced in section 5.7.1, the communication manager uses the device-ioctl function to start the sending routine and the kernel timer. In fact the function is mainly used by the communication manager for controlling the timed-package-send routine. Therefore it is necessary to give more information about this function here.

Function ioctl() is a Linux user space function which is for controlling devices and streams. For controlling a device, the device driver should have a function so that the call to ioctl() is directed to it. The function usually has many functional blocks organized by a switch-case statement. Function ioctl() uses a requesting parameter to indicate the intended functionality of a call. This function also allows an optional parameter as an input data along with the requesting parameter.

The device-ioctl function is the corresponding function that handles the call to ioctl() in the communication manager. In the current implementation, the function has the following features:

1. Starting the sending routine – runs the routine as mentioned in section 5.7.1.
2. Stopping the sending routine – simply let the routine quit the outer loop which is shown in figure 5-8.
3. Starting the kernel timer – it is supposed that the timer starts after the sending routine.
4. Stopping the kernel timer – turns off the kernel timer.
5. Changing the time slot – an important feature that allows the robot agent to adjust the length of the time slots during runtime.
6. Manual send packages – manually send out the package(s) in the current outgoing buffer. This feature is only useful for testing purpose and the sending routine should not be running.

7. Manually change the robot selector – this feature is for testing the sending routine only.

Because the sending routine has to run in a different process from the one that the package-import routine belongs to, it is easier and more flexible to start the routine from a user space task with the ioctl() function. The current scheme is to use a task queue to start the routine as soon as the communication manager is loaded, but it turned out to be difficult to set up and stop properly. One should keep in mind that as long as the sending routine is running, the task is occupied by the routine[25].

5.7. Receiving up-link packages

Just like the DMA transmission in sending down-link packages, the communication manager cannot control and monitor the procedure of the DMA transmission (from rx-FIFO to the incoming buffer) in receiving up-link packages. The receiving-interrupt-handler routine, as an interrupt handler, is only activated after the PC87108A raises the end-of-reception interrupt. The event to be captured for the end-of-reception interrupt is every second level of the **st-FIFO**[26].

Although the current package exchanging scheme between the NWPCS and the NWBOS is in one-request-one-response style, the design of the communication manager tries to prepare for one-request-multi-response style so that the data inside the NanoWalkers can be transmitted to the NWPCS as soon as possible. In the current implementation, IR receiving mode is the default mode of PC87108A. The IR receiving mode can be prepared[27] by three routines under different situations respectively:

25 This is why when testing the communication manager, the routine is always started with a simple program in a Linux console window with an "&" symbol at the end of the command line.

26 One can refer to section 2.4 for more about this kind of interrupt.

27 The preparation includes the DMA channel and rx-FIFO, as well as enabling the interrupt event capturing for IR receiving.

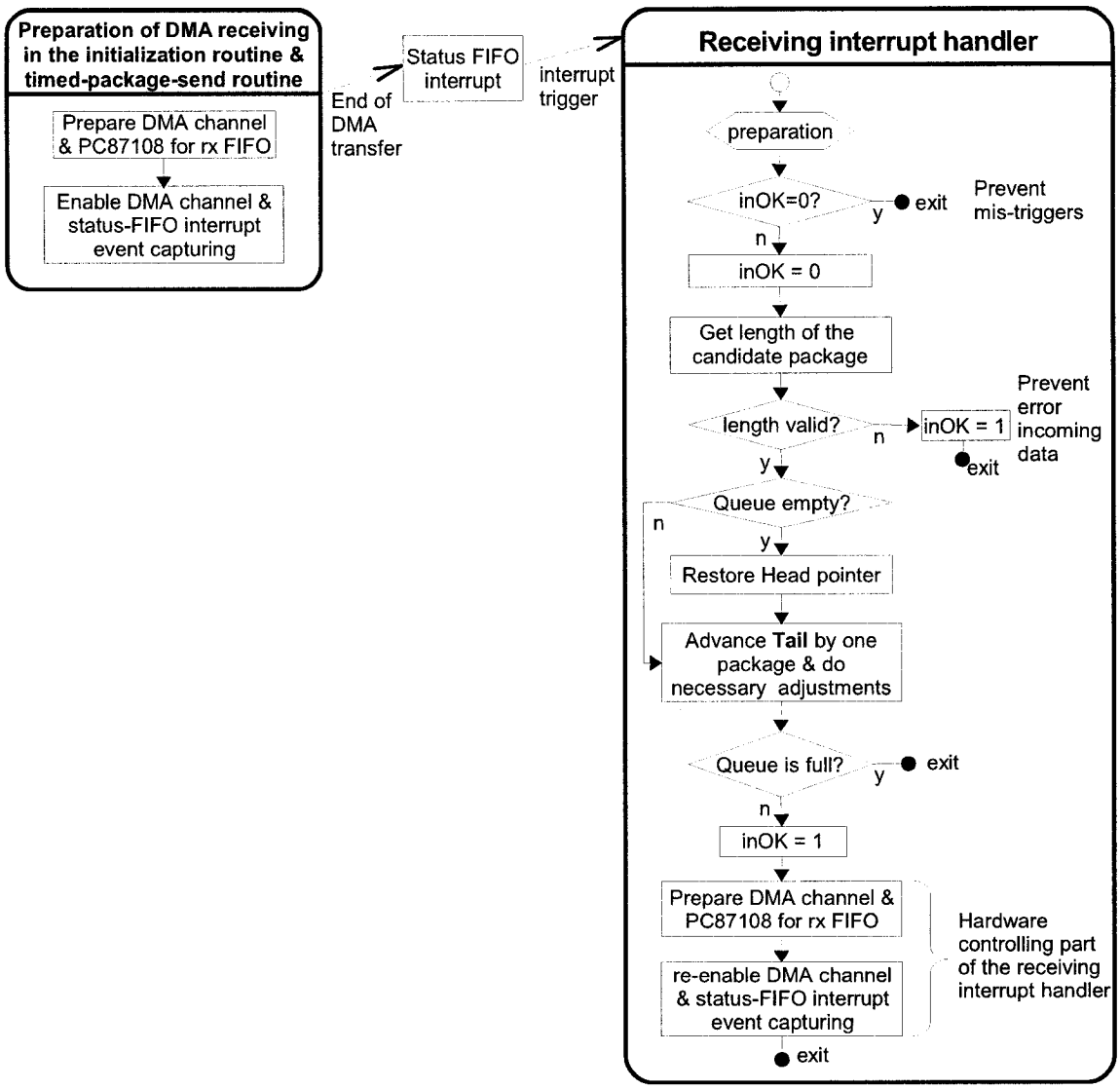

Figure 5-9: The control flow diagram of the receiving-interrupt-handler routine.

1. by the initialization routine, during the communication manager being loaded into the kernel;
2. by the transmission interrupt handler of the timed-package-send routine (as
3. shown in figure 5-8), after the handler wakes up **wait_queue 2**;
4. by the receiving-interrupt-handler itself, after adjusting the **Head**, **Tail**, and **End** pointer to new values.

The control flow diagram of the receiving-interrupt-handler is as shown in figure 5-9, and the detailed algorithm of receiving an up-link package is as the following steps:

1. If the value of inOK is zero, the PC87108A is mis-triggered by some unknown signal source, just ignore the interrupt and exit.
2. Set the value of **inOK** to zero, so that the timed-package-send routine will not send any package before the handler ends.
3. Find out the length of the new up-link package (in the incoming buffer) indicated by its **Package Length field**. If the length is less than eight or greater than the maximum package length, ignore the new package[28], restore the value of **inOK** to one, and exit the handler.
4. Restore **Head** to the same as **Tail** if the queue is empty before receiving the new package, and advance **Tail** to the element right after the end of the new package.
5. If the queue is in forward case, and the space from **Tail** to **End** is less than the maximum package length, set **End** to **Tail** and set **Tail** to the beginning of the chunk.
6. If the queue is in inverse case, and the space from Tail to Head is less than the maximum package length, then the incoming buffer is in full state; the handler should end at this point.
7. If the queue is not in full state, set **inOK** to one.
8. Set up the PC87108A for the next IR reception.
9. Prepare the DMA channel on both the PC computer side and the PC87108A side for transferring data from the rx-FIFO.
10. Re-enable the DMA channel and end-of-reception interrupt capturing.

The last three steps belong to the hardware controlling part of the receiving-interrupt-handler routine, as shown in figure 5-3, because they directly deal with the registers in

28 This is to reduce the possibility of the PC87108A controller being triggered by some unknown data sources. In fact the tests of the implementation showed that this possibility exists.

PC87108A and the DMA controller of the ISA bus.

5.8. The algorithm of importing packages

As mentioned previously, when the robot agent calls the write() function of Linux to write a down-link package to the communication manager, the call is passed by the device-write function to the package-import routine. The robot agent does not care about which robot the communication manager is currently dealing with. The down-link package is supposed to be appended to the end (pointed by the **Tail** pointer) of the circular queue of the target buffer.

The algorithm of how the package-import routine imports the down-link package includes the following steps and figure 5-10 shows the flow chart diagram of the algorithm:

1. Find out the target outgoing buffer indicated by the **NWID** field of the down-link package.
2. Find out the length of the down-link package indicated by the **Package Length field** of the package.
3. Obtain the tx-semaphore of the target outgoing buffer with the **down_interruptible**() function to protect the **Head**, **Tail** and **End** pointers. At that moment if the tx-semaphore is being held by the timed-package-send routine, the package-import routine will be put to sleep until the semaphore is released.
4. Back up the current values of the **Head**, **Tail** and **End** pointers so that they can be restored in case the buffer is full or any error happens later.
5. Restore the **Head** pointer to the value of the **Tail** pointer if the buffer is empty.
6. Check if the buffer is in full state. In forward case, if the space from **Tail** to the end of the chunk is not enough for holding the down-link package, set the **End** pointer to the **Tail** pointer and the set the **Tail** pointer to the beginning of the chunk. In inverse case, if the space from **Tail** to **Head** is not enough for holding the package, then the buffer is in full state, and the routine has to restore the

Head, **Tail** and **End** pointers and return a value to the robot agent indicating that the target buffer is full for the package.

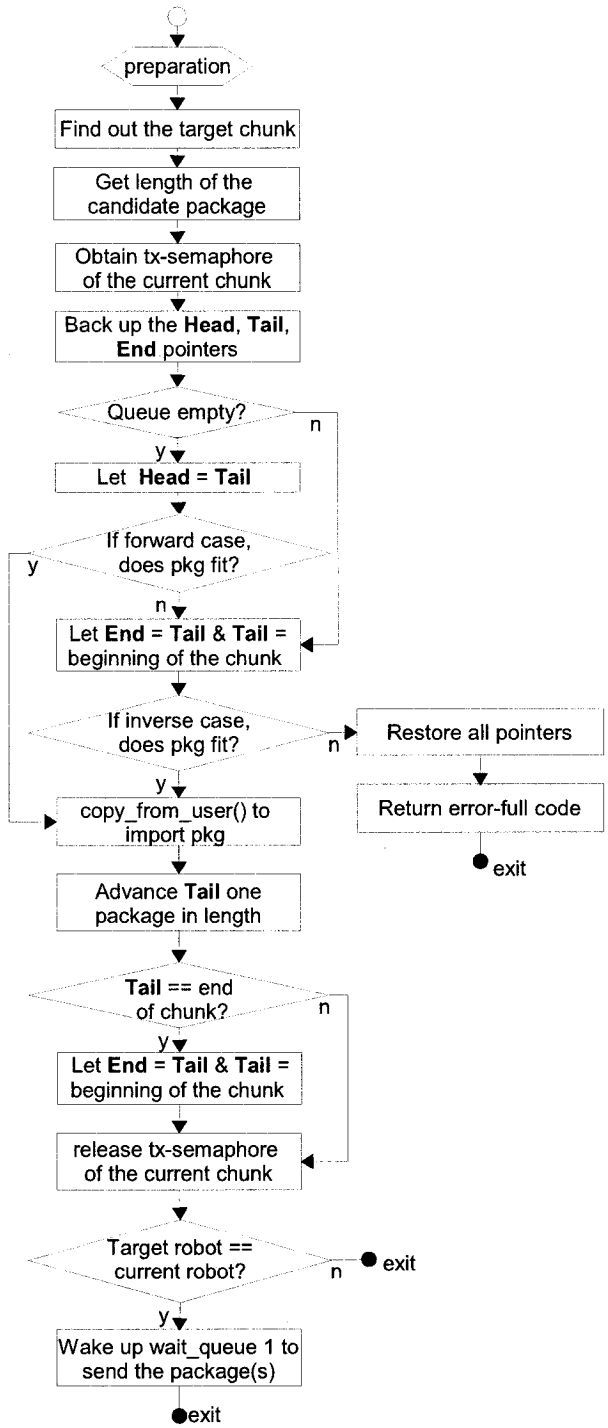

Figure 5-10: Control flow chart of the package-import routine.

7. Use the **copy_from_user**() function to append the content of the package into the space starting at **Tail**. In case the copying fails for some reason, the importing procedure is failed; all pointers are restored and the routine will return an error code to the robot agent.

8. Advance the **Tail** pointer to the element right next to the end of the new package. If the **Tail** pointer is right at the end of the chunk, set the **End** pointer to **Tail** and set **Tail** to the beginning of the chunk.

9. Release the tx-semaphore of the target buffer. If the target buffer belongs to the current robot, wake up the sending routine as mentioned in section 5.7.2.

It is possible that the down-link package is added to the current chunk and the current time slot is not finished yet. In such case the old package(s) in the current chunk have been sent out (otherwise the operation importing the down-link package has to wait for the sending routine to release the tx-semaphore), and the sending routine is put to sleep again. The last two steps of the flow chart in figure 5-10 is to ensure that the newly added package can be sent out within the current time slot.

Another thing worth to be mentioned is that the data stream in and out of the communication manager are always in whole packages. Although, as specified by Linux, the write() function has a parameter (nbyte) indicating the length of the data stream being written into the device, the parameter is ignored by the package-import routine; because the length of the stream can be found in the **Package Length field** of the package. The same situation applies to the read() function too – the stream length parameter (nbyte) of read() is ignored by the package-export routine.

5.9. The algorithm of exporting packages

Similar to importing a package, when the robot agent calls the read() function of Linux to read an up-link package from the manager device, the call is passed by the device-read function to the package-export routine. The candidate package is the one at the beginning

of the queue (pointed by the **Head** pointer) of the incoming buffer. The algorithm of how the package-export routine exports the up-link package includes the following steps:

1. If the incoming buffer is empty, return a value to the robot agent indicating that nothing is exported.

2. Find out the length of the candidate up-link package in the queue. The length is indicated by the Package Length field of the package.

3. Use the **copy_to_user**() function to make a copy of the candidate package to the robot agent. In case the copying fails, the exporting procedure is failed, and an error code is returned to the robot agent.

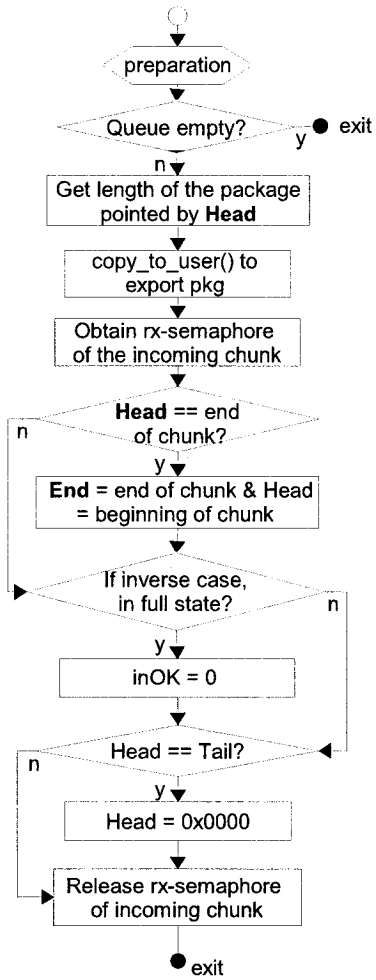

Figure 5-11: Control flow chart of the package-export routine.

4. Now the routine is ready to change the **Head**, **Tail** and **End** pointers as well as

the **inOK** flag. Before doing this, the routine tries to obtain the rx-semaphore of the incoming buffer with the **down**() function. Now adjust the **Head** pointer to the element right after the end of the candidate package. If the queue becomes empty now, **Head** is the same as **Tail**. In such case, let **Head** points to *0x0000*. However, if the queue is not empty yet, here **Head** is at to the beginning of the next up-link package.

5. If the queue is not empty and the **Head** pointer is the same as the **End** pointer after the adjustment, it means that the queue before adjusting the **Head** pointer is in inverse case; this inverse case now ends and the queue should be set to forward case. To set the queue to forward case, the **End** pointer is put back to the end of the chunk, and the **Head** pointer is further set to the beginning of the chunk.

6. Moreover, if the queue is in full state before adjusting the **Head** pointer and now has enough space to hold a package with the maximum package length, set **inOK** flag to one.

7. Release the rx-semaphore so that the receiving-interrupt-handler can continue to work.

The operations from step 5 to step 7 are very simple and very quick. Figure 5-11 shows the flow chart diagram of the algorithm of the package-export routine. The purpose to make processes simple is to minimize the blocking time of the receiving-interrupt-handler, because a very long blocking time may cause the next up-link package sent by the current robot being lost.

5.10. Conclusion and discussion

This chapter has fully described the design of the IR communication manager. Some implementation details that relates to the key design idea, e.g. the distribution of the routines in the three kernel modules, are also mentioned. The whole chapter can be summarized as the following points:

- The IR communication manager is better to be designed as a Linux device

driver so that its specifications can be achieved easily.

- The external structure of the manager is divided into three kernel modules, so that the manager can have better extendibility.
- The internal structure of the IR communication manager consists of many sub-routines. The design idea and the algorithm of each sub-routine are described carefully.
- The manager uses special circular queues to buffer the outgoing and incoming packages so that they can handle large amount of package data and allow DMA transmissions.
- The IR communication manager uses a wait_queue and a kernel timer task to allocate time-slots to the robots.
- The IR communication manager uses semaphores and integer flags to synchronize the major sub-routines.
- The package sending sub-routine is not only triggered by the timer task, but also the package-import routine. It uses another wait_queue and an interrupt handler to synchronize the IR transmission time delay.

Although the test results of the communication channel show that the design of the IR communication manager meets the current specifications of the NanoWalker project, some modifications may be needed in the future. Some of these modifications could be:

- From section 5.3.3 one can see that the current one-send-one-receive package exchanging scheme between the NWPCS and the NWBOS is in fact enforced by the generation of the **DMA Status FIFO** event. The PC87108A controller requires at least two DMA transmissions to generate the expected interrupt. The one-send-one-receive scheme ensures no up-link package being missed by the NWPCS. As a comparison, the TIR2000 controller only needs one DMA transmission to generate a similar interrupt. Therefore if the PC87108A is replaced by another IR communication controller, it is possible that manager will use a one-send-multiple-receive scheme, which may increase the up-link

data transmission rate given the channel bandwidth is not changed.

- From section 5.4 one can see that using the **maximum package length** setting to arrange the circular queue of the incoming buffer is not so efficient, especially when the **maximum package length** setting is pretty large. If the IR communication manager can check the package type and body of the down-link package current being sent out, it may be able to estimate the possible length of the next up-link package, and prepare the incoming buffer based on that estimation. Arranging the buffer in this way will obviously be more efficient in space, but the examination of the packages might take quite some time.

All of these modifications are to enhance the reliability or the efficiency of the IR communication manager. They need to be well tested before being adopted into the implementation.

Chapter 6. The architecture of NWBOS

The NanoWalker basic operation system (NWBOS), as introduced in chapter three, is a program that runs inside the NanoWalkers. This program can be considered as the lowest layer in the layered-architecture of the NWPCS, but in fact it is an independent system. The "basic operation system" is not an "operating system", because it is much simpler and only focused on controlling a set of hardware components inside a NanoWalker directly and with a capability of IR communication. It is more like an "intelligent" device driver that can control a set of sophisticated hardware operations if given some simple commands.

6.1. The general architecture

Figure 6-1 shows the major hardware component relationships of the NWBOS. The wide arrows represent the data flow directions. In figure 6-1, the DSP mainly controls the TIR2000 and the CPLD. It is the CPLD which translates the commands from the DSP and then controls the operations of the STM, the piezo-legs, the positioning LEDs and the temperature sensors. The use of CPLD makes the circuit of a NanoWalker a lot simpler and more efficient. One can refer to the relevant documentation for more details about the design [15].

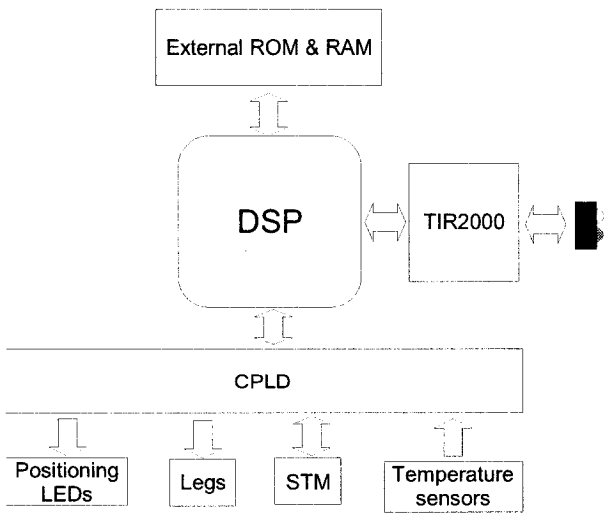

Figure 6-1: The hardware component relationships of the NWBOS.

Figure 6-2 shows the software architecture of the NWBOS. The wide arrows also represent the directions of data flow. Thin arrows represent the utilization of global variables. In figure 6-2, the **main** module is to coordinate the operations of other modules. It accepts the operation requests received by the **IR communication** module, and translates these requests to commands to the **positioning LED control** module, **STM CONTROL & data acquisition** module, **displacement control** module, and **temperature acquisition** module. Then, it invokes the IR communication module again to send out the outgoing data back to the NWPCS. The positioning LED control module is a simple routine that triggers the CPLD to start/stop flashing of the LEDs. Most modules, except the IR communication module, are still under planning or only partially implemented by other students, and are not going to be mentioned in detail.

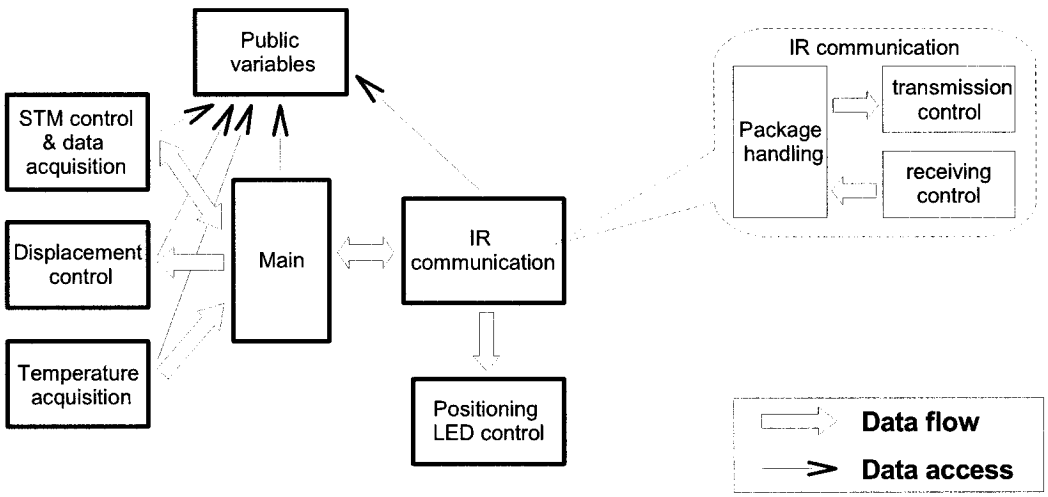

Figure 6-2: The software architecture of the NWBOS.

The IR communication module is the one to be mainly concerned in this chapter. It includes three functional units: the **package handling** unit, the **transmission control** unit, and the **receiving control** unit, as shown in figure 6-2. The transmission control unit and the receiving control unit are concerning about controlling the hardware operations of the TIR2000. The package handling unit is concerning about parsing the received packages into (incoming) data and operation requests of other modules. The outgoing data, including the status data, temperature data and the STM data are converted to packages by the package handling unit automatically without the care of the main module. From the layer

point of point view, the IR communication module is corresponding to the IR communication manager; the transmission control and the receiving control units are corresponding to the device driver. In the actual implementation, all of the three units are put in one program file (IRLib.c), so that they can share some global variables. The receiving control unit and the transmission control unit share the same interrupt handling function for the TIR2000 controller.

The **public variables** module in figure 6-2 collects all data variables that are public to all modules. Typical public data are: one unsigned character of the robot identification number, three temperature data (in an array), a set of locomotion data for legs (in a structure), one operating status data (in an unsigned character), a set of STM CONTROL data (in a structure), a set of STM data properties (in a structure), etc. The structure that holds the locomotion data includes one array of vibration bit sequences (one word each), one word for number of bit sequences, one word of vibration frequency, and one word for number of repetitions (of movement). The structure that holds the STM data includes one pointer to the STM image buffer, the width and height of the STM image buffer, and the current scanning position (reserved for future). Because the current compiler for the DSP does not support malloc() function, the STM image data array can be defined at compilation time with a large enough size. Such an array can be called the **STM data space**. The structure that holds the STM CONTROL data includes the data fields summarized in table 4-9 (all of them are in single word variables).

6.2. The IR communication module

Since every NanoWalker does not know when the NWPCS will send package to it, the NWBOS normally stays in the receiving mode. For every package received, the NWBOS has to reply a package back. The IR communication module is designed to be able to reply a package immediately once it receives a down-link package.

The implementation of the IR communication module is basically finished. The original implementation of this module was done by a student [22]. This version of implementation only had the transmission control and receiving control units, but it did not work properly. The major problems of the version included:

- unable to receive data all the time,
- the message to be sent out cannot be longer than 32 bytes,
- cannot correctly handle messages with odd number of bytes.

In addition, this version used a predefined outgoing buffer to form an outgoing package before sending. This was not necessary and might take a substantial amount of time for copying a long stream of data to the buffer. All of these problems have been fixed. The whole package handling unit has been developed in this research, so that the NWBOS can communicate with the NWPCS for testing the communication channel. The following sections will describe the current version of the three units in detail.

6.2.1. The receiving control unit

The receiving control unit mainly includes a circular queue as the receiving buffer, an interrupt handler including the functions to read the data in the receiving FIFO to the receiving buffer. The receiving buffer is for holding incoming package(s). Unlike the incoming buffer of the NWPCS, this buffer does not have the **End** pointer to represent the dynamic turning point of the circular queue, because the TIR2000 does not use DMA mode for receiving and transmission. The size of the receiving buffer is enough for holding at least one longest possible down-link package. When a down-link frame reaches the TIR2000, the receiving control unit will do the following operations:

- The threshold of the receiving FIFO is set to 56 bytes[29]. Whenever the level of data in the receiving FIFO exceeds the receiving threshold, the TIR2000 raises a "rx-fifo-threshold" interrupt, and the DSP goes into the interrupted mode to read the data from the receiving FIFO to the receiving buffer 56 bytes long.
- When the **End-Of-Frame** (EOF) bit[30] is seen by TIR 2000, it raises an "end-of-frame" interrupt, and the DSP goes into in the interrupted mode to read the rest of the data in the receiving FIFO to the receiving buffer.

29 All IR controller related data and terminologies in this chapter are referred to the TIR2000 user's manual [25].

30 The End-Of-Frame bit is specified by the IrDA specification.

- It is possible that the DSP is too slow to respond to read the data in the receiving FIFO, and causes an over-flow in the FIFO. In that case part of the incoming data will be lost, and the down-link package is ignored.
- As mentioned before, the DSP always works on words rather than bytes. It reads the receiving FIFO two bytes at a time. If the down-link package is in odd number of bytes, an unsigned character 0xff is appended to the last byte of the package. The 0xff can be used for checking if the reception operations are correct during debugging.
- Finally, the receiving control unit sets the **reception status flag** (a global variable) to **true** to indicate that there is a new package in the receiving buffer.

The NWBOS is a single-task system. The main() function, which is in the main module, is in general an endless loop. It continuously checks the reception status flag to see if there is any new down-link package received or not. Once the **reception status flag** is **true**, it will call the package handling module to analyze the incoming package, and give a corresponding package back.

6.2.2. The sending control unit

The sending control unit is initiated by the package handling unit. Unlike the receiving control, the current version of sending control unit does not have a long outgoing buffer. Instead, it has a 4-word package header buffer, which is for making up the header of the outgoing package. Other components of the sending control unit include one function for starting the transmission logic of TIR2000, and one interrupt handler with a function to write the outgoing data to the transmission FIFO.

The threshold of the transmission FIFO is set to 64 bytes. Whenever the level of data in transmission FIFO is lower than the transmission threshold, the TIR2000 generates a "tx-fifo-threshold" interrupt, and the interrupt handler writes 64 bytes of outgoing data to the transmission FIFO. One can see in section 6.1 that the outgoing data, i.e. the temperature data, the STM data and the operating status data, are in arrays. While writing a package to the transmission FIFO, the interrupt handler uses a **current-sending-position** pointer to

scan through the outgoing data arrays word by word, firstly the array of package header, secondly the array of temperature data, finally the status data or a piece of STM data. At the end of writing the package, the interrupt handler sets an **end-of-transmission** (EOT) bit to terminate the transmission.

The advantage of sending a package this way is that the sending control unit does not need to spend a lot of time to copy outgoing data to a specific buffer, and thus also does not need to allocate a big chunk of memory for such a buffer.

6.2.3. The package handling unit

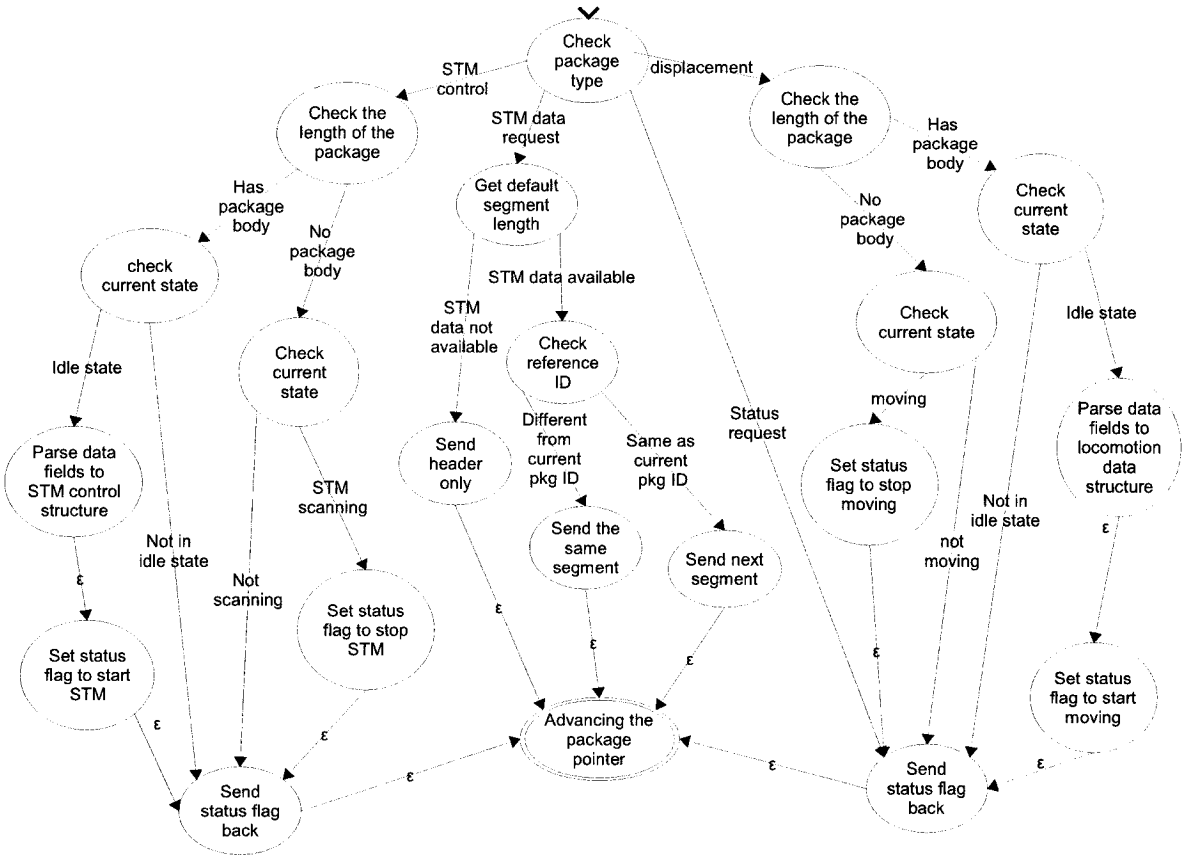

Figure 6-3: The finite state machine for parsing incoming packages.

The package handling unit parsing an incoming package is like a language compiler parsing a program statement. It has a lot of conditional branches while analyzing the package. Figure 6-3 shows the finite state machine of the package handling unit which parsing a down-link package. In figure 6-3, the transits commented with character **ε** are **unconditional** transits, or alternatively are the **default** transits. The starting state is "check

package type", which checks the value of the **Message Type** field. The final state is "advancing the package pointer", which advances the **Head** pointer of the incoming buffer to the next incoming package (if available). A NanoWalker is at idle state if it has no operations. In the current design, if the robot is not scanning and not moving, then it is idling. Overall, the finite state machine can be explained as the following:

- The package handling unit first checks the package type of the incoming package;
- If the package is a STM CONTROL type, check the length of the package.
  - If the length shows that the package has no body, this package is requiring stopping the STM scanning. If the STM is scanning, set the status flag to "STM stop" and send this status flag back to the NWPCS. The STM scanning routine will continuously check this flag during scanning, and quit whenever "STM stop" is detected. After the STM scanning quits, the NWBOS goes to the idle state;
  - If the package has a body, this is a normal STM CONTROL package. If the robot is at idle state, retrieve all controlling data to the corresponding variables, set the status flag to "STM start", and send this status flag back to the NWPCS. The main() function continuously checks the status flag at idle state, and starts STM scanning whenever "STM start" is detected.
- Similar to the STM CONTROL type, if the package is DISPLACEMENT type, check the length of the package.
  - If the package has no body, and the robot is moving, set the status flag to "Moving stop", and send the status flag back to the NWPCS. The displacement routine will stop the movements once the "Moving stop" is detected, and the NWPCS goes to idle state.
  - IF the package has body, and the robot is at idle state, retrieve all locomotion data to the corresponding variables, set the status flag to "Moving start", and send this status flag back to the NWPCS. The main()

function starts the movements whenever "Moving start" is detected.

- If the package is STM DATA REQUEST type, get the expected segment length from the body of the package.
  - If there is no STM data in the buffer, just send the header of a STM DATA package back to the NWPCS.
  - If the STM data is available, then check the reference package ID of the incoming package. It is possible that the previous STM data segment is not received successfully by the NWPCS and the latter requests this segment again. In that case, the reference package ID of the incoming package is different from the current local package ID, and the package handler should send the previous STM data segment again.
  - If the STM data is available and the reference package ID is the same as the current local package ID, send the next STM data segment of which the maximum length is as specified by the incoming package.
- If the package type is STATUS REQUEST, simply send back the status flag.

Finally, the package handler advances the "current incoming package" pointer to the position of the next incoming package.

6.2.4. Synchronization of positioning LEDs

The previous chapters have mentioned that the GSPS uses PSDs to detect the position of robots to determine their positions. Each robot has two positioning LEDs. The GSPS detects the flashing light of the LEDs to determine their positions. The PSD is a pointer source detector. The LEDs have to flash one by one so that the PSD can have accurate outputs. The problem is: how to let the LEDs flash one by one, and how to let the GSPS know which LED flashes at which moment?

The easiest way to solve the problem is to let the IR communication to synchronize the flashing of LEDs. The algorithm is as the following steps:

1. At the beginning of any time slot, the NWPCS sends a down-link package to the target robot. When the robot receives the down-link package, it begins to flash the first LED, and at the meantime, it responds an up-link package to the NWPCS. When the NWPCS receives the up-link package, it invokes the position manager to detect the first LED, and provides the latter the target NanoWalker ID.

2. Later the NWPCS sends the second down-link package to the target robot. When the robot receives the second package, it begins to flash the second LED and responds the second up-link package to the NWPCS. When the NWPCS receives the second up-link package, it tells the position manager to detect the second LED.

3. Finally, the NWPCS sends the third down-link package to the target robot. When the robot receives the package, it shuts off the second LED and responses the third up-link package to the NWPCS. And when the NWPCS receives the third up-link package, it tells the position manager to shut off the detection of the second LED and calculate the position of the target robot.

With respect to the positioning LED control module, the control logic is simple: it uses an integer to count the number of receptions. The counter starts from zero; for every incoming package, the counter increases one, and will be set back to zero when grows to three. Then, if counter is one, trigger the first LED; if two, trigger the second; if zero, shut off the LED.

The benefit of such an algorithm is the simplicity. The drawback of it is that it needs at least six package transmissions in each time slot, and the performance of the system may be relatively low. For example, using the concepts in section 7.1, if the average turnaround time of a down-link package is 7ms in the system, then transmitting six packages requires 7ms×(6packages/2) = 21ms; and if there are 100 robots in the system, the allocation cycle will be at least as long as 0.021second×100 = 2.1 seconds. With the 2.1-second turnaround time, the NanoWalker system may seem to be too slow to be a high throughput real-time system.

Fortunately, if the NanoWalker system has multiple working-cells, it is unlikely that all robots will gather in one working-cell frequently. Therefore each communication manager does not have to allocate time-slots to the robots that are not in its working-cell, and the average allocation cycle of each communication manager can be shortened.

6.3. Conclusion and discussion

This chapter describes the architecture design of the NWBOS, and gives a detailed description about the design and implementation on the IR communication module. As a counterpart of the IR communication manager in the NWPCS, the IR communication module is structurally similar to the latter; and both of them are parts of the IR communication channel.

The whole NWBOS is written in C with Texas Instrument Code Composer II. The tests on the IR communication channel (mentioned in the chapter seven) have proved that the implementation of the IR communication module is working as desired. However some modifications may need to be done in the future. Two examples of these modifications are:

- Allowing robots to communicate in a multi-working-cell system – current design is only for single working-cell.
- Making the allocated STM data buffer as a big circular queue if the IR communication can be done during STM scanning.
- Improve the structural independence of the NWPCS to the DSP hardware architecture so that it can easily adapt to a newer version of DSP in the next generation of NanoWalker system.

The design of the IR communication can be also affected by some unknown factors. For example, the design of locomotion control module and STM scanning control module are not started yet, how they can cooperate with the status flag is not yet clear; how to instruct the CPLD to trigger the flashing of LEDs is unknown too. These problems need to be solved in the future.

Chapter 7. Testing the communication channel

The previous two chapters have described the design of the IR communication manager and the IR communication module of the NWBOS. This chapter describes the tests of the whole IR communication channel. The test is to determine the following factors:

- The reliability of the system
- The shortest possible time slot in the system
- The average turnaround time of the system

The first two factors represent the two important issues described in the beginning of chapter five. The third factor is to find out how to improve the IR communication channel. These three factors are tested one by one, and will be described later.

To test the communication channel, a robot agent simulator need to be developed, which simulates the IR communication controlling part of the robot agent. The simulator program includes two parts: one is running as a package receiver which reads the IR communication manager regularly; the other is running as a package sender that sends one or more down-link packages to the IR communication manager. The actual features of the simulator are subject to be changed for testing different factors. On the NanoWalker side, a DSP circuit board had been made by previous student(s) [15] for running the NWBOS. The circuit board includes the controlling circuit of the TIR2000 controller.

7.1. Some key concepts used in this chapter

In addition to the background knowledge about the Linux kernel, some other key concepts need to be explained before describing the tests of the communication channel.

7.1.1. The allocation cycle

The concept of **allocation cycle** is used with regard to the communication manager. The allocation cycle is the time period between one time slot and the next time slot that are allocated to the same robot, as shown in figure 7-1. In the current design, the time slots are

fixed in length (variable length is too complicated to be considered in the current stage). The allocation cycle

$$T_{cycle} = T_{slot} \times N_{robot}, \qquad (7\text{-}1)$$

where T_{slot} is the length of a time slot, and N_{robot} is the total number of robots in the NanoWalker system or the maximum number of robots that can be put in a working-cell, whichever is the less.

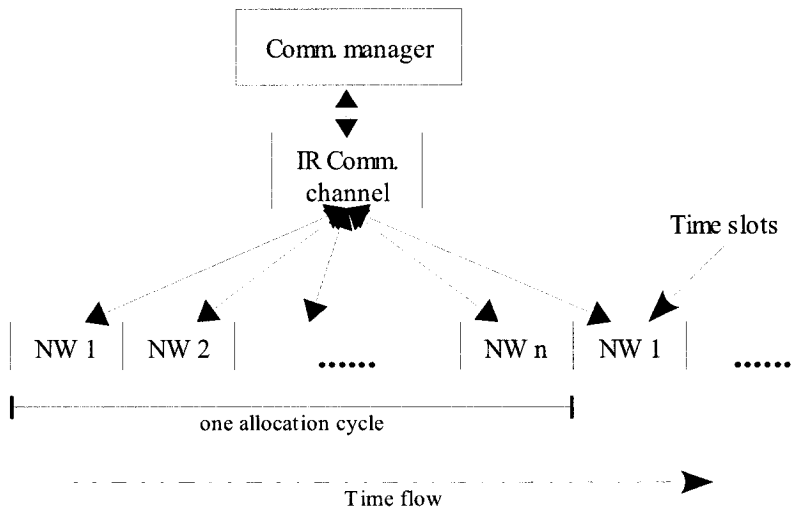

Figure 7-1: Channel allocation and an allocation cycle.

The allocation cycle is an important parameter of the NanoWalker system. It is one of the important factors that determine how frequent the robot agent can update the data of all the robots and thus how well the NanoWalker system is acting as a real-time controlling system.

7.1.2. The time slot

The time slot is used with regard to the communication manager. As one can see in formula 7-1, T_{slot} is the real parameter that needs to be improved. It relates to the bandwidth of the FIR protocol, and the speed of all the software components in the IR communication channel that handle the transmission of the packages. More specifically, the length of T_{slot} is roughly determined by

$$T_{slot} \geq (N_{PCSendins} \times V_{CPU} + T_{IRsend} + N_{DSPrecvins} \times V_{DSP}) \times N_{down\text{-}link} + (N_{PCRecvins} \times V_{CPU} + T_{IRrecv} + N_{DSPsendins} \times V_{DSP}) \times N_{up\text{-}link} \qquad (7\text{-}2)$$

Where

- $N_{PCSendins}$ is the maximum number of CPU instructions of the manager that is used for sending a package (from the outgoing buffer);
- $N_{PCRecvins}$ is the maximum number of CPU instructions of the manager that is used for receiving a package (into the incoming buffer of the communication manager);
- V_{CPU} is the average CPU speed (seconds per instruction) of the computer on which the communication manager is running;
- T_{IRsend} is the maximum transmission time for the IR communication hardware components (TIR2000 and PC87108A and their supporting circuit) to pass a package to a NanoWalker;
- T_{IRrecv} is the maximum transmission time for the IR communication hardware components to pass package back to the CCMS;
- $N_{DSPsendins}$ is the maximum number of DSP instructions the DSP sends out a package;
- $N_{DSPrecvins}$ is the maximum number of DSP instructions the DSP receives in a package;
- V_{DSP} is the average operating speed (seconds per instruction) of the DSP;
- $N_{down\text{-}link}$ is the desired number of down-link package during the time slot
- $N_{up\text{-}link}$ is the desired number of up-link package during the time slot.

Among the above parameters, the T_{IRsend} and T_{IRrecv} are determined by FIR channel bandwidth (4Mb/s), the speed of the IR controllers, and the maximum length of down-link and up-link packages respectively. V_{CPU} and V_{DSP} depend on the hardware characteristics only. $N_{PCSendins}$, $N_{PCRecvins}$, $N_{DSPsendins}$, and $N_{DSPrecvins}$ are the ones needed to reduce for enhancing time efficiency. In practical it is hard to really count the total number of CPU instructions of the program modules. Instead one can focus on finding algorithms that seem to have least

CPU operations. $N_{down\text{-}link}$ and $N_{up\text{-}link}$ are parameters decided by the designer of the system. In the current design, $N_{down\text{-}link}$ always equals to $N_{up\text{-}link}$. In such case, $N_{down\text{-}link}$ or $N_{up\text{-}link}$ should be decided when the value of the shortest time slot is maximum. Here, the shortest time slot is defined as

$$T_{IRsend} + T_{IRrecv} + (N_{PCSendins} + N_{PCRecvins}) \times V_{CPU} + (N_{DSPsendins} + N_{DSPrecvins}) \times V_{DSP} \quad (7\text{-}3).$$

Figure 7-2 shows how the parameters in formula 7-3 relate to the time distributions within a shortest possible time slot. Such a definition ignores any delays caused by other processes running in the operating system.

For the communication manager, proper length of the time slots is critical to the performance of sending and receiving packages: on one hand the system wants the time slots to be as short as possible to enhance the performance; on the other hand, since the 4Mb of the communication channel bandwidth is not a very high rate, to allow communicating with a large number of robots, the accuracy of allocating the time slots to each robot is very important. For example, suppose that the NanoWalker system wants all of the robots to have a chance to communicate with the NWPCS every second so that the

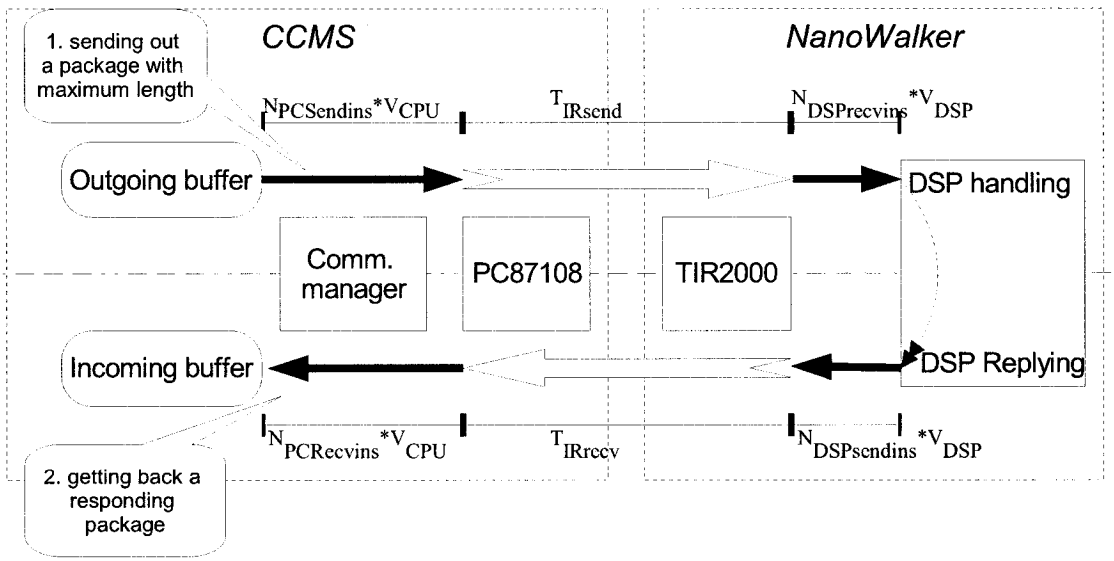

Figure 7-2: The minimum estimation of a time slot.

performance of the system resembles to real time. If there are 100 robots in the working-cell, then the time slot is only 10ms wide. Because the task of receiving the communication

packages is done with DMA, which is not controllable by the time slot allocation process, the IR communication manager has to make sure that it has finished receiving the desired package from the current robot[31] before the current time slot ends.

In the implementation of the communication manager, the length of the time slots is defined as a command line parameter which can be changed when the manager is loaded into the kernel.

7.1.3. The turnaround time

For the robot agent, the turnaround time of a down-link package is the time from sending out this package (to a communication manager), to getting its responding (up-link) package back (from a communication manager), as shown in figure 7-3. In a multi-working-cell environment, there are multiple communication managers, and if the robot agent sends a package to one communication manger, it can get the responding package from another communication manager. Anything (the device driver, the robots, etc) below the manager layer is transparent to the robot agent.

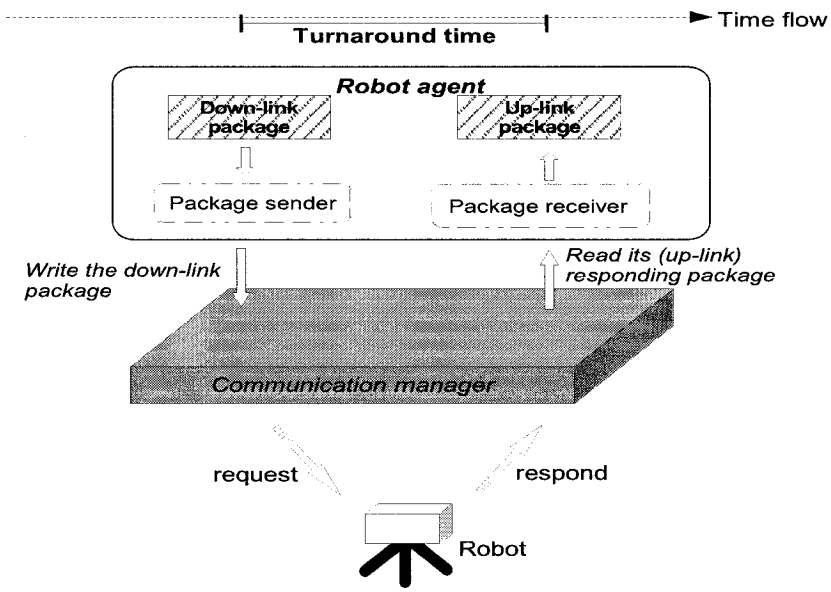

Figure 7-3: The turnaround time of a down-link package.

The turnaround time is variable from one down-link package to another. What are meaningful are the **average turnaround time** and the **maximum turnaround time**. From

31 **Current robot** means the NanoWalker that the current time slot is allocated to. The same meaning of this term applies to the rest of the thesis.

communication point of view, the 4Mb/s of FIR specification is the transmission rate of the IR communication hardware (or device driver); the reciprocal of the allocation cycle is the transmission rate of the communication manager; the reciprocal of the average turnaround time is the average observed transmission rate of the communication channel; and the reciprocal of the maximum turnaround time is the minimum observed transmission rate of the communication channel. The maximum turnaround time occurs when the down-link or/and up-link packages are long, and sending and receiving of packages are not within the same time slot.

The turnaround time depends not only on the time slot length of the communication managers and the number of robots in the system, but also on the time that the robot agent spends on writing down-link packages to the outgoing buffers of the communication manager(s) and reading up-link packages from the incoming buffers of the communication manager(s). In a time-sharing operating system, these portions of time consumptions are hard to be calculated. Therefore in practical, one can only get the average turnaround time and maximum turnaround time by experiments.

7.1.4. The combinational package length

Sending a down-link package is always followed by receiving an up-link package. Observing the time or other resource consumption on either way is not practical. The time or other resource consumption has to be observed from the beginning of sending a package to the end of receiving the responding package. Thus it is more meaningful to use the combinational package length (the sum of the down-link package length and the up-link package length) concept to express the length of the package streams in the experiments for this thesis.

7.2. Testing the reliability

The reliability of the IR communication manager means no synchronization problem, no missing package or partially received package. It is not only related to the algorithm of the manager, but also the internal runtime parameters of the manager, such as the length of the packages, the size of the buffers, length of the time slots, etc.

7.2.1. Testing methodology

The reliability test is an important test which should be based on large number of package transmissions. For each type of down-link packages, the simulator sends out 10 thousand example packages and verifies the responding up-link packages received. Among the up-link packages, the length of the responding STM data package bodies varies within the range of 0, 10, 100, 500, 800, 1000, and 1600 bytes. For each down-link package, its Local Package ID should be the same as the Reference Package ID of its corresponding up-link package; the contents of the up-link package body should be as expected as well.

Because there is only one DSP circuit currently available, the number of target robot is only one.

7.2.2. Results and analysis

Although the total testing time was pretty long, the testing results showed that all up-link packages were correctly received within the allocation cycle. The channel therefore is as reliable as expected.

Comparatively, the experiment results obtained by another student [23] showed that the packages with lengths less than 304 bytes had no problem for transmissions, but packages longer than 304 bytes always failed. The problem was caused by the simulator of that experiment which used the functions in the standard C++ **iostream** class to read and write the IR communication manager. As mentioned in chapter five, the simulator should use the file handling functions provided by Linux.

7.3. Estimating the shortest possible time slot

Finding out the shortest possible time slot is important for evaluating the performance of the channel, and hence the performance of the whole system. Since most of the parameters in formula 7-2 are hard to be calculated, the shortest possible time slot can only be estimated with experiments here.

7.3.1. Experiment methodology

Because the minimum time clock interval is 10ms in Linux, and the shortest time slot could be less than 10ms, to get an accurate estimation, the experiment runs the same sending and receiving operations, as depicted in figure 7-2, for many times. The total time consumption is recorded. Then the average shortest possible time is the total time consumption divided by the number of sending and receiving operations. Please note that the sending and receiving operations mean the operations from the beginning of the timed-package-send routine to the end of the receiving-interrupt-handler routine only.

In this experiment, there is no time slot allocation needed, because the loops for testing the total time consumption cannot be broken. Only one robot (the DSP circuit prototype) is involved.

7.3.2. Results and discussion

Figure 7-4 shows the testing results of the (average) total time consumption versus the combinational package lengths ranging within 23, 100, 200, 300, 900 and 1500 bytes. The number of sending and receiving operations is ranging within 1000, 2000, 3000 and 4000 loops. This figure gives the following conclusions:

- The total time consumption proportionally increases with the number of loops (sending and receiving operations), which means that there is basically no extra operations (of the computer) to do as the number of loops increasing during the tests.

- The shortest possible time slot increases proportionally with the combinational length of the packages. From the point where the combinational package length is 23 bytes to the point where the combinational package length is 300 bytes, the curves of all test results are basically straight lines. The turning of the curves at point 300 is only because the scale of the plot is changed. The curves from point 300 to point 1500 are also straight lines. These straight lines mean that the total time consumptions is proportional to the combinational package length.

- Unlike expected, if packages are short, the major limitation on the actual

bandwidth of the IR communication channel does not come from the bandwidth of the FIR, but comes from the processing speed of the programs in the PC computer of the CCMS and the DSP of the NanoWalker. The shortest possible time slot is about 2ms at the point where the combinational package length is 23 bytes. The 2ms time period approximately equals to the general processing time for sending and receiving packages. Referring to formula 7-3, the general processing time for sending and receiving packages is corresponding to the terms other than T_{IRsend} and T_{IRrecv}.

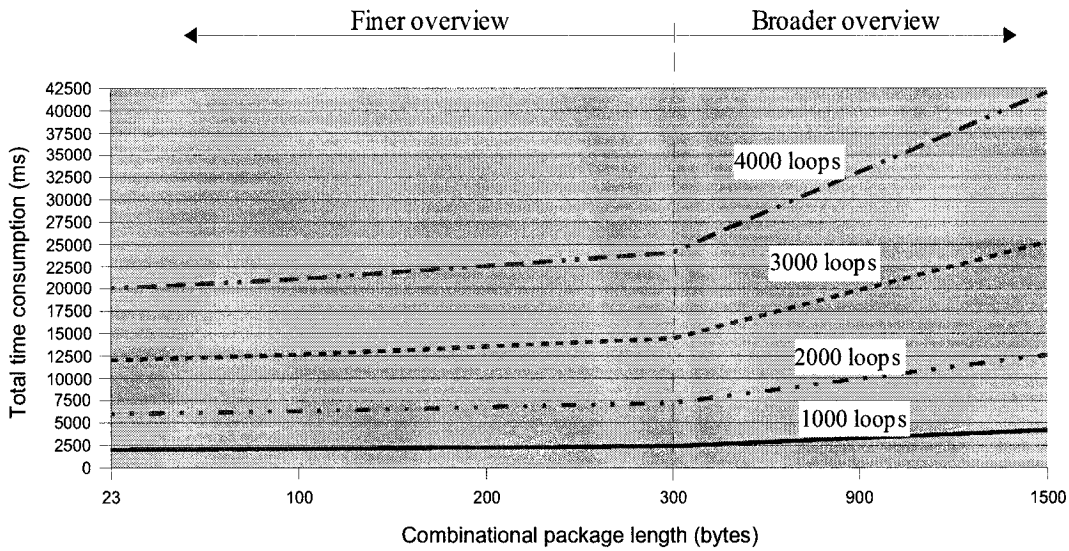

Figure 7-4: The total time consumption vs. the length of packages

To be able to synchronize the positioning LEDs of the robots and the PSD, the actual time slot of the system should allows at least three sending-receiving of packages (totally six packages) to be processed. Therefore, if the maximum package length is 1500 bytes, for example, the actually time slot of the system should be more than $4.22 \times 6 \approx 25$ms.

In addition, to increase the actual channel bandwidth, it is important to optimize the algorithms of the timed-package-send routine and the receiving-interrupt-handler routine in the IR communication manager and the IR communication module in the NWBOS. Optimizing the algorithms can certainly reduce the processing time of the routines. A primitive test had shown that the processing speed of the PC87108A and TIR 2000 controllers are much slower than the processing speed of the CPU of the PC computer and the DSP. Thus reducing the number of operations on these controllers (in the hardware

controlling routines) can significantly reduce the total processing time. Using faster hardware components are also important. Faster hardware components (CPU, memory, IR controller, DSP, etc.) can directly reduce the total processing time of the programs.

7.4. Finding out the average turnaround time of the system

The average turnaround time is also another important factor for evaluating the performance of the system. If the sending and receiving of packages occur within the same time slot, the turnaround time relates to the length of the shortest possible time slot, the processing time of the device-write routine, device-read routine, the read() function in Linux, the write function in Linux, and the scheduling period under the current operating system status. The scheduling period may increase if the number of tasks in the operating system increases [23]. If the sending and receiving of packages occur in different time slot, then the turnaround time also relate to the length of the allocation cycle. Due to all of the factors mentioned here, the changes of the package length will have less effect on the average turnaround time than on the shortest possible time slot.

7.4.1. Experiment methodology

As in the experiment of estimating the shortest possible time slot, the experiment here also uses many loops to get more accurate total time consumptions. The results of total time consumption divided by the number of loops will give the average turnaround time of the system. For simplicity, the number of loops is fixed at 1000.

Although there is only one DSP circuit board available, to allow various allocation cycles, the experiment assumes that there are "virtual NanoWalkers" in the system. No down-link packages are written to those "virtual NanoWalkers", so there is no sending and receiving operations happen when time slots are allocated to them.

7.4.2. Results and discussion

Figure 7-5 shows the results of the average turnaround time vs. the combinational length of packages for a single robot system. Here, the combinational package length is ranging within 23, 100, 200, 300, 900 and 1500 bytes. Figure 7-6 shows the results of the

average turnaround time vs. the number of robots in the system. Here, the number of robot is ranging within 1, 10, 50, 100, and the combination length of packages is 23 bytes. The figures give the following conclusions:

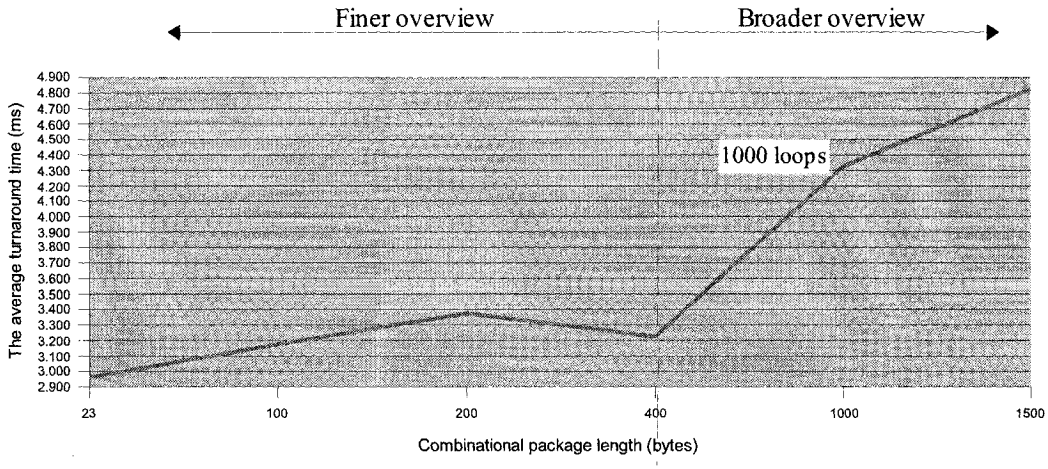

Figure 7-5: The average turnaround time vs. the combinational length of packages.

- In figure 7-5, it is obvious that the turnaround time increases as packages getting longer, but does not have obvious affect when the length of packages is short. This means that the major portion of the turnaround time is consumed by the NWPCS and NWBOS, and is not very much affected by the 4Mb/s IR transmission rate. To reduce the average turnaround time, one should reduce the shortest possible time slot and the processing time of other routines in the NWPCS and NWBOS.

- In figure 7-5, the average turnaround time at point 400 and point 1500 has lower value than they ought to be. Such a phenomenon appears all the time during the tests. That may be because that Linux uses different algorithms to handle different sizes of data copying in function copy_to_user() and function copy_from_user().

- In figure 7-6, the average turnaround time increases proportionally with the number of robots. This is expectable because the more robots are in the system, the longer the allocation cycle will be.

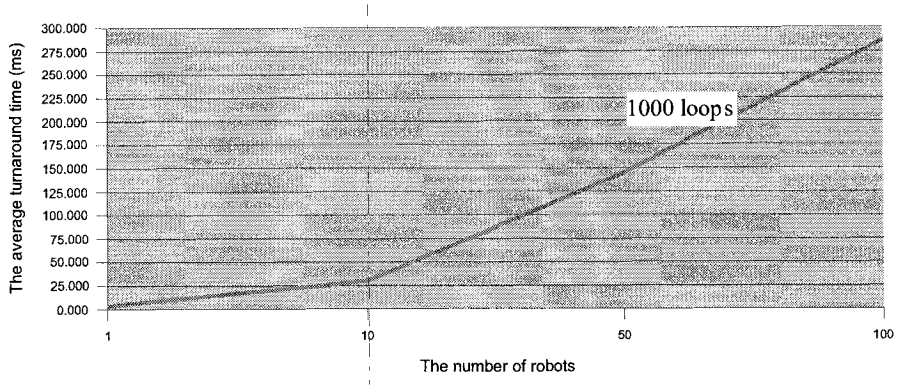

Figure 7-6: The average turnaround time vs. the number of robots.

After all, the average turnaround time of the system can be found from figure 7-5 as long as the combinational package lengths are known. Usually, the system is more concerning about the maximum value of the average turnaround time. This value can be checked from the figure at the point where the combinational package length is the maximum package length plus 10 (the length of an STM DATA REQUEST type package).

7.5. Conclusion

The testing results show that the IR communication channel is reliable for exchanging packages. The length of the shortest possible time slots increases as the package getting longer. So does the length of the actual time slots. As the allocation cycle increases as the time slot increases, the performance of the whole system will decrease.

The problem of the current IR communication channel is that the shortest possible time slot is too long for a high efficiency system. Such a long time slot is not because of the 4Mb/s IR transmission rate, but mainly because of the algorithms of the routines in the NWPCS and NWBOS, as well as the hardware components in the CCMS and the NanoWalkers. This means that optimization on the modules in the IR communication channel may be necessary.

In general, the average turnaround time of the system also increases as the length of packages increases. However, because Linux uses different algorithms to handle different size data blocks, the average turnaround time may decrease a little bit as the value of the package lengths increases to some points.

Chapter 8. Conclusion and future works

This thesis describes the design of the controlling system architecture of the NanoWalker project, as well as the development (design and implementation) of the infrared communication channel for the project. It intends to provide a solid knowledge base to the future developers of the NanoWalker system to proceed on the software controlling system.

8.1. Conclusion

The thesis first describes modifications in the design of the layered architecture. An original version of layered architecture had been proposed by previous students. The architecture had six layers as: the hardware layer, the driver layer, the manager layer, the agent layer, the NanoOS layer, and the application layer. Data exchanged between layers were done through inter-layer communication links. While the original layered architecture had drawbacks that the roles of some layers were not clearly defined, the inter-layer communication concept was not proper for the actual development, and the hardware layer definition was not necessary.

A new layered architecture is proposed in this thesis which clarifies the roles of all layers, the components in each layer and the logical relationships among those components. The new proposal supports multi-working-cell environment. In the new proposal, the hardware layer is replaced with a new layer, the NWBOS layer; the inter-layer communication concept is replaced with the inter-computer communication concept. The inter-computer communication uses RPC as the data exchange protocol.

The NanoWalkers communicate with the central controlling and management platform with infrared communication channels. Within a channel, the data streams are divided into packages. Detailed structure of the packages and the exchanging protocols are described in the thesis. The thesis then describes the detailed structure and implementation of the IR communication manager.

The IR communication manager uses a set of circular queues to hold the incoming and

outgoing packages. The communication channel is allocated (by the communication manger) to each robot one by one for a short period of time. The short period is called time slot. The processes of sending and receiving packages of the IR communication manager need to be synchronized so that the packages can be exchanged in a one-send-one-receive scheme.

On the NanoWalker side, there is also a program routine which is the counterpart of the IR communication manager. The detailed structure and implementation of the routine is also described in this thesis.

Finally, the communication manager channel is tested to make sure of its reliability. The shortest possible time slot and the average turnaround time of the system, two important factors that evaluates the performance of the system, are also checked and analyzed.

8.2. Future works

To enhance the performance of the system, the first thing that can be done is to modify some hardware components in the system: try to use faster computers, use faster DSP, and the IR communication controllers, replace the FIR protocol with the newer VFIR protocol (that require the replacement of the IR communication controllers too), replace the version of Linux to the newer versions that has faster clock rate and better algorithms of multi-tasking. The second thing is to refine the algorithms of the routines in the IR communication manager. The third thing is to extend the IR communication manager to support multiple working-cells.

The integration of the position manager with the IR communication manager is not yet finished, and needs to be continued. The design of the robot agent needs to be continued too.

For the new generation of NanoWalker system which is currently under planning, many hardware components will be changed, so do the software components that monitor or/and control them. Any designs of the components in the NWPCS should think about these components can be reused in the next generation system.

References or Bibliography

[1] Alan H. Goldstein. "Everything you always wanted to know about nanotechnology...", http://www.salon.com/tech/feature/2005/10/20/nanotech/index.html, Oct. 20, 2005.

[2] Michael P. and Judy Dong. "Diameter of an Atom", *The Physics Factbook*, http://hypertextbook.com/facts/MichaelPhillip.shtml,1996.

[3] S. Martel, P. Madden, L. Sosnowski, I. Hunter, and S. Lafontaine. "NanoWalker: a fully autonomous highly integrated miniature robot for nano-scale measurements", *Proc. of the European Optical Society (EOS) and SPIE International Symposium on Envirosense, Microsystems Metrology and Inspection*, Vol. 3825, Munich, Germany, 12 pages, June 14-18, 1999.

[4] S. Martel, Olague L. Cervera, Bautista Coves Ferrando J., S. Riebel, T. Koker, J. Suurkivi, T. Fofonoff, M. Sherwood, R. Dyer, and I. Hunter. "General description of the wireless miniature NanoWalker robot designed for atomic-scale operations", *Proceedings of SPIE: Microrobotics and Microassembly*, Vol. 4568, pp. 231-240, Oct. 29-31, 2001

[5] S. Martel, M. Sherwood, J. Suurkivi, and I. Hunter. "An infrastructure suited for supporting a fleet of wireless miniature robots designed for atomic-scale operations", *Proceedings of SPIE: Microrobotics and Microassembly*, Oct. 29-31, 2001.

[6] S. Martel and G. Baumann. "Infrared positioning and communication unit for a nanorobotics platform operating in a cold helium atmosphere," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IEEE/RSJ IROS 2003)*, Oct. 27-31, 2003

[7] Crommie Condensed Matter Research Group, University of California, Berkeley, Material

Science Division, LBNL. "Scanning Tunneling Microscope", http://www.physics.berkeley.edu/research/crommie/research_stm.html

[8] S. Martel, T. Koker, and I. Hunter. "Main design issues for embedding onto a wireless miniature robot, a scanning tunneling positioning system capable of atomic resolution over a half-meter diameter surface area", *Proceedings of SPIE: Microrobotics and Microassembly*, Oct. 29-31, 2001.

[9] S. Martel, M. Sherwood, C. Helm, W.G. de Quevedo, T. Fofonoff, R. Dyer, J. Bevilacqua, J. Kaufman, O. Roushdy, and I. Hunter. "Three-legged Wireless Miniature Robots For Mass-scale Operations At The Sub-atomic Scale", *Proc. of the 2001 IEEE International Conference on Robotics and Automation (ICRA), vol. 4, pp. 3423-3428*, May 2001.

[10] S. Martel, M.; O. Roushdy, M. Sherwood, and I. Hunter. "High-resolution optical positioning system for miniature instrumented", *Proc. SPIE: Microrobotics and Microassembly II*, Vol. 4194, pp. 121-128, Nov. 5-6, 2000.

[11] Patrick J. Megowan, David W. Suvak, and Charles D. Knutson. "IrDA Infrared Communications: An Overview". Counterpoint Systems Foundry, Inc.

[26] "Wikipedia, the free encyclopedia", http://en.wikipedia.org/wiki/Kalman_filter.

[12] Grant Wilson. "OSI Model Layers", http://www.geocities.com/SiliconValley/Monitor/3131/ne/osimodel.html, August 6, 2001.

[13] Dominic St-jacques. "Conception D'une Architecture Logicielle Pour Le Positionnement Au Niveau Atomique D'instruments Scientifiques Sous Forme De Robots Miniatures Pour Des Applications En Nanotechnologies", master's thesis, Département De Génie Électrique, École Polytechnique De Montréal, 2004.

[14] Texas Instruments. "TMS320 Second-generation Digital Signal Processors", http://focus.ti.com/lit/ds/symlink/tms320c25.pdf, 1990.

[15] Marc-Antoine Fortin. "Alimentation Intermittente D'un Robot Miniature Et Sans Fil, Synchronisée Avec Ses Déplacements", master's thesis, Département De Génie Informatique, École Polytechnique De Montréal, 2005.

[16] Sylvain Martel, Juan Bautista Coves Ferrando, Lorenzo Cervera Olague, Timothy Fofonoff, and Ian Hunter. "Implementing frequency modulated piezo-based locomotion for achieving further miniaturization for wireless robots". Proceedings of SPIE: Microrobotics and Microassembly, pp. 210-220, Newton, MA, Oct. 29-31, 2001.

[17] Fares Jaradeh. "Système de positionnement à l'échelle globale d'une flotte de nano robots", PFE project, Sep, 2005.

[18] Dominic St-Jacques, Thomas Boitani, Pierre-Alain Dumas, Marc-Antoine Ducas, Marc-Antoine Fortin, and Sylvain Martel. "Atomic-Scale Positioning Reference Grid System for Miniature Robots with Embedded Scanning Tunnelling Capability", *Proceedings of the 2004 IEEE, International Conference on Robotics & Automation, New Orleans, LA*, April 2004.

[19] Dave Marshall. "Remote Procedure Calls (RPC)", http://www.cs.cf.ac.uk/Dave/C/node33.html, May, 1999.

[20] Alessandro Rubini & Jonathan Corbet. "LINUX Device Driver", 2nd . édition, O'REILLY, June, 2001.

[21] Ngakeng Chimi Serge. "The infrared communication device driver", PFE project, Apr, 2003.

[22] Marc Léger. "Programmation du DSP TMS320C25-50 embarqué sur le robot

NanoWalker", PFE project, Oct, 2004.

[23] Hong Jun Song. "IMPLEMENT NANOWALKER AGENT AND ROBOT SIMULATOR", research report for Master (course) project, 2005.

[24] National Semiconductor. "PC87108AVHG/PC87108AVJE, Advanced UART and Infrared Controller", user's manual, August, 1998.

[25] Texas Instruments. "TIR2000 Data Manual", user's manual, June 1998.

[26] Open Network Computing Remote Procedure Call Working Group. "ONC Remote Procedure Call (oncrpc)", http://www.ietf.org/html.charters/OLD/oncrpc-charter.html, 2001.

Appendixes

Appendix A The interfaces of the robot agent

As one can see in figure 3-2, the robot agent is an important component which controls the core hardware components in the NanoWalker system. Although the design of the robot agent is not yet finished, the interface structure of the robot agent with other layers can be approximately outlined here.

As mentioned in chapter three, the robot agent abstractly represents the current working states of all NanoWalker robots. It has three major duties:

1. storing the all of the instant data of the robots, including the internal temperatures, position coordinates and orientation, acquired STM data, operating status, etc;
2. collecting the above-mentioned data automatically;
3. providing functions for the NanoOS to access those data, and give controlling

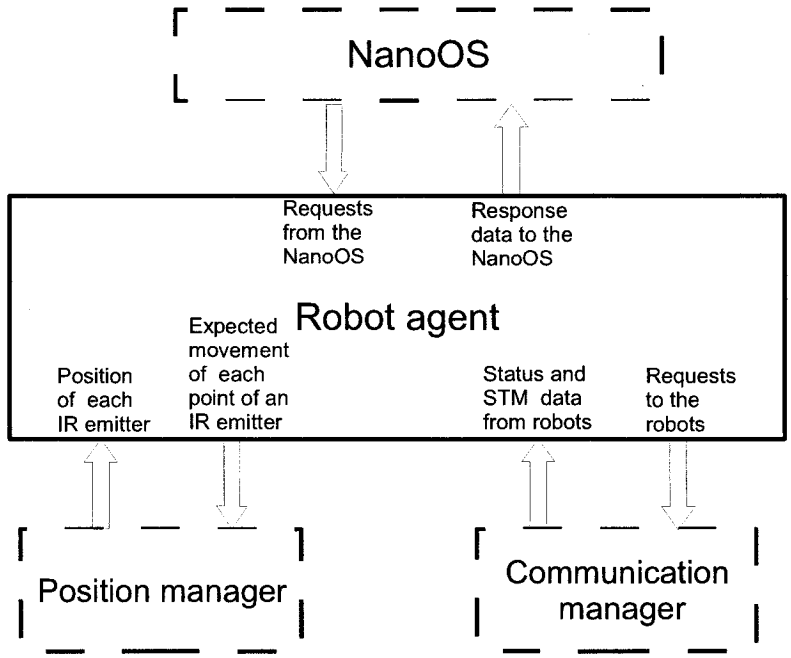

Figure 8-1: The interfaces of robot agent with components in other layers.

commands to the communication manager(s) and the position manager(s).

Figure A-1 shows the input/output data passing through the interfaces between the robot agent and the components in other layers need to be introduced here, especially the data changed with the communication manager(s). How to handle these data determines the internal features of the robot agent.

With regard to the interfaces with the components in the lower layer, the robot agent needs to collect the coordinates of every positional LED from the position manager to calculate the position and orientation of each robot. During the detection of each LED, the agent need to provide the expected position of the LED to the position manager for calibrating the obtained results, as will be mentioned in the next section. Similarly, the robot agent needs to collect other data about the robots from the communication manager, and it need to provide to the robots various requesting data to control their operations.

With regard to the interface with the upper layer, the robot agent should be able to accept management requests from the NanoOS and return any required data about the robots back.

Appendix B About the position manager

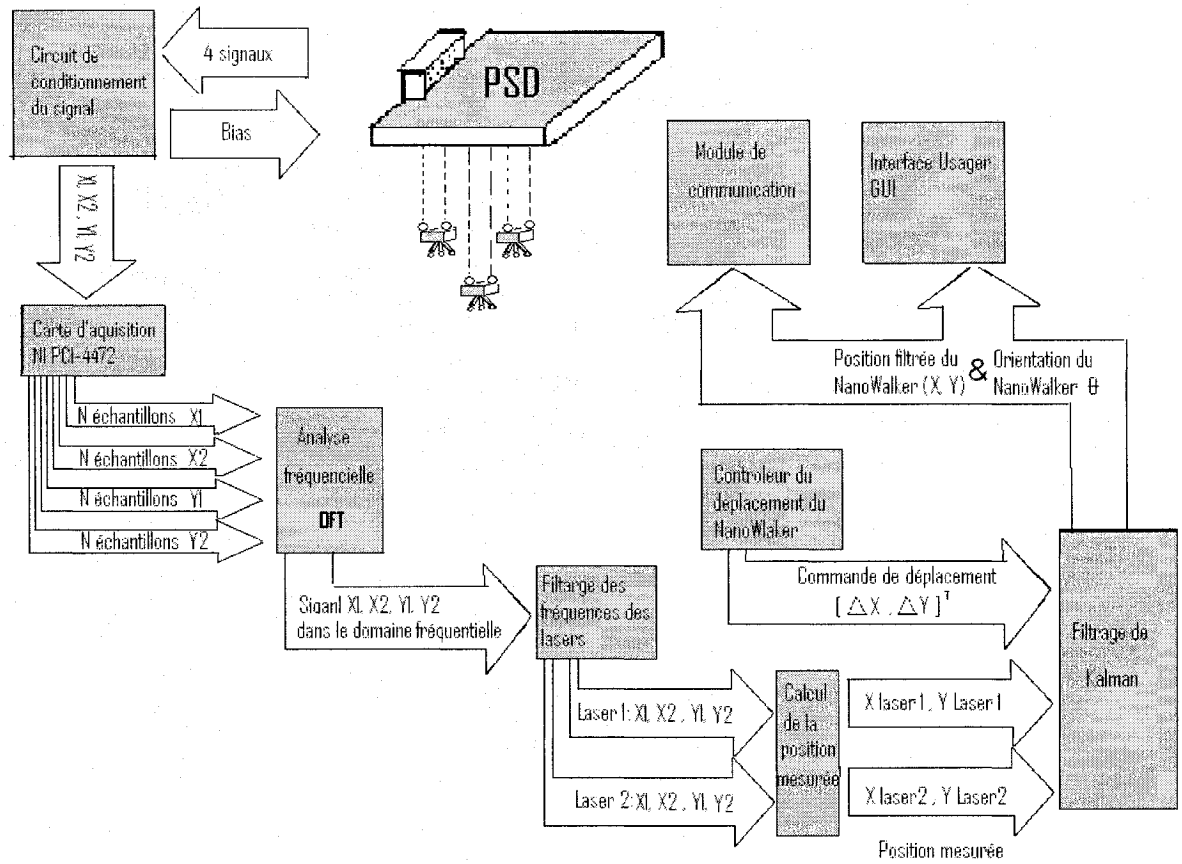

Figure 8-2: The schematic of the position manager [17].

The position manager is in fact a Kalman filter [26] which outputs "most possible" two-dimension coordinates of the infrared positioning emitters of the robots. Unlike the photo sensing devices such as the cameras, a PSD is a highly sensitive device for detecting point source light. Its output always represents the coordinates of a single point of light. In other word, in each sampling, the PSD can only detect the position of one positioning LED. The Kalman filter (position manager) needs to sample many[32] times from the PSD in order to determine an amended position of an IR positioning (within a small range). Since robots may be moving during the sampling, the position manager requires that the robot agent provides the expected moving information of the current (being sampling) robot positioning LED, so that the expected position values at each sampling point can be calculated.

32 The current setting is 200 samples which is for testing the maximum precision of the Position manager only. It takes too much time for practical use. A proper number will be determined by integrated tests.

The position manager had been worked on by many students. The most recent work on the position manager part was refining the program of the Kalman filter so that it could response faster [17] and have better resolution. Figure B-1 shows the block diagram of the position manager prototype used for testing the position manager. The prototype used a laser instead of the IR LED as the signal source. The best resolution of such a prototype was about 20μm. However, on average, the resolution was about 75μm.